Formal Dynamic Semantics of AVA 95

Michael K. Smith

Technical Report: 112 September 1995

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views, conclusions and modifications contained in this document are those of the authors and should not be interpreted as representing the official policies either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Chapter 1
INTRODUCTION

This report contains the operational semantics for the AVA subset of Ada 95. The informal description of this subset can be found in the Reference Manual [Smith 95].

1.1 Notation

Here is a summary of the syntax used in this document in terms of the official syntax of the Acl2 logic.

1. Variables. *x*, *y*, *z*, etc. are printed in italics.
2. Function application. For any function symbol for which special syntax is not given below, an application of the symbol is printed with the usual notation; e.g., the term `(fn x y z)` is printed as fn (*x*, *y*, *z*). Note that the function symbol is printed in Roman. In the special case that 'c' is a function symbol of no arguments, i.e., it is a constant, the term `(c)` is printed merely as c, in small caps, with no trailing parentheses. Because variables are printed in italics, there is no confusion between the printing of variables and constants.
3. Other constants. **t**, **f**, and **nil** are printed in bold. Quoted constants are printed in the ordinary syntax of the ACL2 logic, in a 'typewriter font.' For example, **`'(a b c)`** is still printed just that way. **`#b001`** is printed as $\texttt{001}_2$, **`#o765`** is printed as $\texttt{765}_8$, and **`#xa9`** is printed as $\texttt{a9}_{16}$, representing binary, octal and hexadecimal, respectively.
4. `(if x y z)` is printed as

   if *x* **then** *y* **else** *z* **fi**.
5. `(cond (test1 value1) (test2 value2) (t value3))` is printed as

   if *test1* **then** *value1* **elseif** *test2* **then** *value2* **else** *value3* **fi**.
6. `(case x (key1 answer1) (key2 answer2) (otherwise default))` is printed as

   case on *x*: **case** = `key1` **then** *answer1* **case** = `key2` **then** *answer2* **otherwise** *default* **endcase**.
7. `(let ((var1 val1) (var2 val2)) form)` is printed as

   let *var1* **be** *val1*, *var2* **be** *val2* **in** *form*.
8. `(let* ((var1 val1) (var2 val2)) form)` is printed as

   let* *var1* **be** *val1*, *var2* **be** *val2* **in** *form*.
9. `(forall (x y) (p x))` is printed as

   $\forall$ *x*, *y*: p (*x*).

10. `(exists (x y) (p x))` is printed as

    ∃ *x*, *y*: p (*x*).

11. `(not x)` is printed as

    ¬ *x*.

12. The remaining symbols that are printed specially are described in the following table.

ACL2 Syntax	Conventional Syntax
`t`	**t**
`f`	**f**
`nil`	**nil**
`(lt x y)`	$x <_n y$
`(le x y)`	$x \leq_n y$
`(gt x y)`	$x >_n y$
`(ge x y)`	$x \geq_n y$
`(union-theories x y)`	$x \cup y$
`(set-difference-theories x y)`	*x less y*
`(intersection-theories x y)`	$x \cap y$
`(congruent x y)`	$x \cong y$
`(or x y)`	$x \vee y$
`(and x y)`	$x \wedge y$
`(* x y)`	$x \times y$
`(- x y)`	$x - y$
`(+ x y)`	$x + y$
`(union x y)`	$x \cup y$
`(remainder x y)`	x **mod** y
`(/ x y)`	x / y
`(iff x y)`	$x \leftrightarrow y$
`(implies x y)`	$x \rightarrow y$
`(append x y)`	x @ y
`(member x y)`	$x \in y$
`(>= x y)`	$x \geq y$
`(> x y)`	$x > y$
`(<= x y)`	$x \leq y$
`(< x y)`	$x < y$
`(lessp x y)`	$x < y$
`(greaterp x y)`	$x > y$
`(geq x y)`	$x \geq y$
`(leq x y)`	$x \leq y$
`(e0-ord-< x y)`	$x <_\varepsilon y$

ACL2 Syntax	Conventional Syntax
`(equal x y)`	$x = y$
`(= x y)`	$x =_n y$
`(eql x y)`	$x =_a y$
`(eq x y)`	$x =_{eq} y$
`(not (member x y))`	$x \notin y$
`(not (equal x y))`	$x \neq y$
`(not (= x y))`	$x \neq_n y$
`(not (eql x y))`	$x \neq_a y$
`(not (eq x y))`	$x \neq_{eq} y$
`(minus x)`	$-x$
`(1+ x)`	$1+x$
`(zerop x)`	$x \cong \mathbf{0}$
`(numberp x)`	$x \in \mathbf{N}$
`(1- x)`	$x-1$
`(not (numberp x))`	$x \notin \mathbf{N}$
`(top-as x)`	$x_{\mathrm{a}[1]}$
`(top-vs x)`	$x_{\mathrm{v}[1]}$
`(top-es x)`	$x_{\mathrm{e}[1]}$
`(clock x)`	x_t
`(as x)`	x_{a}
`(vs x)`	x_{v}
`(es x)`	x_{e}
`(length x)`	$\|x\|$
`(len x)`	$\|x\|$
`(abs x)`	$\|x\|$

A superscript "*" indicates repetition and may be used to indicate a list of components, $decl^*$, or a function that acts on a list of arguments, $I_e^*(l, env)$

Multiple values may be set or returned. We use $\langle x, ..., z\rangle$ to indicate such cases.

Lists are composed using square brackets, e.g. [*x*, `1`, [*y*, *z*]].

Lists are composed using square brackets, e.g. [*x*, `1`, [*y*, *z*]]. Literal symbols and lists are quoted with "’", e.g. ’**constraint-error**, ’[*a*, *b*, *c*].

Most semantic operations have two components, an exception check and a modification to the environment. In the ACL2 logic we capture this notion using functions that return multiple values. We present a sequence of such forms by an ‘ilet’ indicated by **{** *form** **}**. An ‘ilet’ returns two values, an exception and an environment.

An ’ilet’ has a procedural flavor, even though an n element ’ilet’ simply expands into an n element-deep nested structure of **`let`**’s. The ‘ilet’ subforms are required to return a pair of values consisting of an exception and a new environment, which are bound to the variables, *exc* and *env*, respectively. The only exception to this is the ‘‘:=’’ operation.

‘ilet’ == **{** *form** **}**

form == ⟨*a*, *b*⟩ |
  exception (*pred*, *[fail]*) |
  check-assert (*pred*, *[out]*, *[in]*) |
  check-asserts (*[out]*, *[in]*) |
  args := *form*; |
  expr
args == *symbol* | ⟨ *symbol** ⟩

In an ‘ilet’ context, these forms are interpreted as follows.

A simple *expr* must return two values which are bound to ⟨*env*, *exc*⟩ before the next form is interpreted.

exception(*form*, *fail*) evaluates *form*. If not **t** in the current *env*, then we continue, otherwise we exit the ‘ilet’ with the multiple value, *fail*. *Fail* defaults to ⟨*form*, *env*⟩.

check-assert(*form*, *[instate]*, *[outstate]*) interprets *form* with respect to the *instate* and *outstate* value stacks. If **t**, then we continue, otherwise we exit the ‘ilet’ with the exception, **logical-error**.

check-asserts(*[instate]*, *[outstate]*) checks the elements of the current assertion stack with respect to the *instate* and *outstate* value stacks. If all are **t** then we continue, otherwise we exit the ‘ilet’ with the exception, **logical-error**.

assert1(*form*) adds *form* to the assertion stack.

asserts(*forms*) appends *forms* to the assertion stack.

a := *form* binds *a* to the value returned by *form*.

⟨*a*, ... *z*⟩ := *form* does a multiple-value bind of ⟨*a*, ... *z*⟩ to the values returned by *form*.

If after the evaluation of any element of **{** *a* ... *b* **}**, the variable *exc* becomes non-false, we return ⟨*exc*, *env*⟩. For this reason you will sometimes see the exception compontent of a form bound to *exc2* so that we can handle it explictly in the semantics.

1.2 Notes on the Implementation

Mutual recursion is difficult to reason about. So, we avoid it whenever possible.

The environment, *env*, consists of three stacks of stacks: the entry stack, the value stack, and the annotation stack.

The entry stack holds declarations (of objects, subprograms and packages). The value stack holds the results of expression evaluation. The annotation stack is intended to handle the accumulated requirements due to transition and invariant assertions.

Structure of the environment:

env	= [entry-stack value-stack assertion-stack]
entry-stack	= **nil** | [local-estack . entry-stack]
local-estack	= **nil** | [entry . local-estack]
value-stack	= **nil** | [local-vstack . value-stack]
local-vstack	= **nil** | [*value* . local-vstack]
assertion-stack	= **nil** | [local-astack . assertion-stack]
local-astack	= **nil** | [*lexpr* . local-astack]
entry	= [*id* ttype *value]* | [*id decl]*

Chapter 2

OPERATIONAL DEFINITION

We begin with the top level definition of the interpeter.

SET CURRENT PACKAGE to be **ACL2**.

INCLUDING the book: *ava-dynamic*.

INCLUDING the book: *predefined-packages*.

In ‘interpret-eval’ *form* is the form to be interpreted, with *flg* indicating what type of form it is. We use a single large recursive function because mutual recursion is difficult to reason about in ACL2.

flg = ’DECL for a declaration,
flg = ’DECLS for a list of declarations,
flg = ’STMT for a single statement,
flg = ’STMTS for a list of statements,
flg = ’EXP for an expression,
flg = ’EXPS for a list of expressions,

The *env* is composed of three parts:

- env_v is a stack of value stacks.
- env_a is a stack of annotation stacks.
- env_e is a stack of entry stacks, which includes at the top level the various predefined procedures and types.

The clock variable, *c*, in the interpreter is the maximum stack depth of subprogram calls. In addition to normal procedure and function recursion, each cycle of a loop is counted as a subprogram call.

DEFINITION:
I_{flg} (*form*, *env*, *c*)
=
if ¬ pos-int-p (*c*) **then** hard-error-env (*env*, `"Out of time"`, *c*)
elseif env-p (*env*)
then case on *flg*:
  case = `stmts` **then**
    if consp (*form*)
     then { I_s (car (*form*), *env*, *c*)
        I_s^* (cdr (*form*), *env*, *c*) **}**
     else ⟨**nil**, *env*⟩
    fi
  case = `exps` **then**

```
   if consp (form)
    then {  I_e (car (form), env, c)
            I*_e (cdr (form), env, c)  }
    else ⟨nil, env⟩
   fi
case = decls then
   if consp (form)
    then {  I_d (car (form), env, c)
            I*_d (cdr (form), env, c)  }
    else ⟨nil, env⟩
   fi
case = stmt then
   if consp (form) ∧ top-prefix-p (form, nil)
    then case on statement-opr (form):
           case = sl then  I*_s (arg* (form), env, c)
           case = constrained-st then
              {  pre := env;
                 I_s (constrained-st-stmt (form), push-as ([constrained-st-relation (form)],
                                                            env), c)
                 ⟨exc, pop-as (env)⟩
                 check-asserts (env_e, pre_e)
                  }
           case = null then  ⟨nil, env⟩
           case = assign then
              let* var be assign-var (form),
                   value be assign-value (form),
                   root be root* (var)
                   in
              {  pre := env;
                 ⟨exc, env1⟩ := I*_e (actions (var), push-env, c – 1);
                 actions := env1_v[1];
                 env2 := env;
                 I_e (var, env, c)
                 I_e (value, push-vs (nil, env2), c)
                 value := top-value (env);
                 env := pop-vs (env);
                 assign-to-env (root, actions, value, env)
                 check-asserts (env_e, pre_e)
                  }
           case = proc-call then
              let* proc-name be proc-call-id (form),
                   proc be proc-lookup (proc-name, env),
                   formals be arg* (procedure-params (proc)),
                   actuals be arg* (proc-call-actuals (form)),
                   spec be      procedure-spec (proc)
                            ∨  '(true)
                   in
              {  pre := env;
                 I*_e (actuals, push-env (env), c)
                 I*_d (reverse (formals), env, c – 1)
                 instate := env;
                 ⟨exc2, env⟩ := I_s (procedure-body (proc), env, c – 1);
                 exception (null (exc2),
                              hard-error-env (env,
```

```
                                   "Procedure exit with NULL exc"))
      exception (exc2 ≠ *subprogram-return*,
                    ⟨exc, pop-env (env)⟩)
      exc := nil;
      outstate := env;
      check-assert (spec, outstate, instate)
      I*_e (extract-ids (formals), env, c − 1)
      values := reverse (env_v[1]);
      env3 := pop-env (env);
      ⟨exc2, env⟩ := I*_s (assign-actuals (formals,
                                            actuals,
                                            values), pop-env (env), c − 1);
      exception (exc2, ⟨exc2, env3⟩)
      check-asserts (env_e, pre_e)
      }
case = return then
   if return-value (form)
    then {  I_e (return-value (form), env, c)
              ⟨*subprogram-return*, env⟩  }
    else ⟨*subprogram-return*, env⟩
   fi
case = exit then  ⟨*loop-exit*, env⟩
case = raise then  ⟨*program-error*, env⟩
case = if-stmt then
   let ifarms be arg* (form)
       in
   if atom (ifarms)  then ⟨nil, env⟩
    else let ifarm be car (ifarms)
             in
         {  pre := env;
            I_e (ifarm-test (ifarm), env, c)
            if      top-value (env)
               =    true
             then I_s (ifarm-statements (ifarm), pop-value (env), c)
             else I_s (mk-if-stmt (cdr (ifarms)), pop-value (env), c)
            fi
            check-asserts (env_e, pre_e)
             }
   fi
case = while-loop then
   {  pre := env;
      I_e (while-loop-test (form), env, c)
      exception (top-value (env) = false,
                    ⟨nil, pop-value (env)⟩)
      env := pop-value (env);
      ⟨exc2, env⟩ := I*_s (arg* (while-loop-statements (form)), env, c);
      exception (exc2 = *loop-exit*, ⟨nil, env⟩)
      exception (exc2, ⟨exc2, env⟩)
      check-asserts (env_e, pre_e)
      I_s (form, env, c − 1)
      }
case = block then
   {  pre := env;
      ⟨nil, push-es (nil, push-as (nil, env))⟩
      ⟨exc2, env⟩ := I*_d (arg* (block-decls (form)), env, c);
```

```
                  exception (exc2, ⟨exc2, pre⟩)
                  ⟨exc2, env⟩ := I_s^* (arg* (block-body (form)), env, c);
                  exception (    exc2
                              ∧  block-handler (form)
                              ∧  handler-error-p (exc2),
                              I_s^* (arg* (block-handler (form)), env, c))
                  exception (exc2, ⟨exc2, pop-es (pop-as (env))⟩)
                  ⟨nil, pop-es (pop-as (env))⟩
                  check-asserts (env_e, pre_e)
                  }
              otherwise hard-error-env (env,
                                        "Undefined statement type!",
                                        form)
            endcase
      else hard-error-env (env,
                           "Unexpected atomic statement!",
                           form)
    fi
case = exp then
    if literal-p (form)  then ⟨nil, push-value (form, env)⟩
     elseif id-p (form)
     then {  value := entry-value (variable-lookup (form, env));
             exception (¬ value,
                        ⟨hard-error2 (nil,
                                      "Unbound variable ~p0",
                                      form), env⟩)
             ⟨nil, push-value (value, env)⟩  }
     elseif indexed-component-p (form)
     then {  ⟨nil, push-env (env)⟩
             I_e (indexed-component-root (form), env, c)
             I_e (indexed-component-index (form), env, c)
             exception (get-array-elem-exc (nth-vse (1, env),
                                            nth-vse (0, env)))
             ⟨nil, push-value (get-t (nth-vse (1, env), nth-vse (0, env)),
                               pop-env (env))⟩  }
     elseif selected-component-p (form)
     then {  ⟨nil, push-env (env)⟩
             I_e (selected-component-root (form), env, c)
             ⟨nil, push-value (get-t (nth-vse (0, env),
                                      selected-component-field (form)),
                               pop-env (env))⟩  }
     elseif aggregate-p (form)
     then hard-error-env (env,
                          "Unqualified aggregate",
                          form)
     elseif aggregate-choice-p (form)
     then hard-error-env (env,
                          "Unqualified choice aggregate",
                          form)
     elseif aggregate-pos-p (form)
     then I_e (aggregate-pos-value (form), env, c)
     elseif qualified-p (form)
     then if aggregate-p (qualified-value (form))
             then let agg be qualified-value (form),
                      typ be qualified-type (form)
                      in
                 if pos-aggregate-p (agg)
```

```
                then { ⟨nil, push-env (env)⟩
                       I_e^* (extract-agg-values (arg* (agg)), env, c)
                       labels := type-tree-labels (typ, nil);
                       ⟨nil, push-value (pairlis$ (labels,
                                                   reverse (env_v[1])),
                                         pop-env (env))⟩ }
                elseif choice-aggregate-p (agg)
                then { ⟨nil, push-env (env)⟩
                       I_e^* (extract-agg-values (arg* (agg)), env, c)
                       labels := type-tree-labels (typ,
                                                   extract-agg-labels (arg* (agg)));
                       ⟨nil, push-value (pairlis$ (labels,
                                                   reverse (env_v[1])),
                                         pop-env (env))⟩ }
                else hard-error-env (env,
                                     "Aggregate must be uniform",
                                     form)
                fi
           else { I_e (qualified-value (form), env, c)
                  exception (coerce-to-subtype-exc (top-value (env),
                                                    qualified-type (form),
                                                    nil))
                  ⟨nil, push-value (coerce-to-subtype-val (top-value (env),
                                                           qualified-type (form)),
                                    pop-value (env))⟩ }
           fi
elseif type-convert-p (form)
then { exception (type-convert-exc (type-convert-value (form),
                                    type-convert-type (form)))
       ⟨nil, push-value (type-convert-val (type-convert-value (form),
                                           type-convert-type (form)),
                         env)⟩ }
elseif op-expr-p (form)
then let opr-name be id-root (op-expr-id (form)),
         operands be arg* (op-expr-actuals (form))
         in
     if opr-name =_eq 'if
      then { ⟨exc, env2⟩ := I_e (operands_0, push-vs (nil, env), c);
             if top-value (env2) = true
              then I_e (operands_1, env, c)
              else I_e (operands_2, env, c)
             fi }
      else { ⟨exc, env2⟩ := I_e^* (operands, push-vs (nil, env), c);
             exception (eval-opr-exc (opr-name,
                                      reverse (env2_v[1])))
             env := push-value (eval-opr-val (opr-name,
                                              reverse (env2_v[1])),
                                env); }
     fi
elseif function-call-p (form)
then let* func-name be function-call-id (form),
          func be func-lookup (func-name, env),
          formals be arg* (function-params (func)),
          actuals be arg* (function-call-actuals (form)),
          spec be function-spec (func) ∨ '(true),
          calling-env be env
```

```
              in
          {  I*_e (actuals, push-env (env), c)
             I*_d (reverse (formals), env, c − 1)
             instate := env;
             ⟨exc2, env⟩ := I_s (function-body (func), env, c − 1);
             exception (null (exc2),
                         hard-error-env (env,
                                        "Function exit with NULL exc"))
             exception (exc2 ≠ *subprogram-return*,
                         ⟨exc2, calling-env⟩)
             outstate := env;
             check-assert (spec, instate, outstate)
             ⟨nil, push-value (top-value (env), calling-env)⟩
              }
    else hard-error-env (env,
                         "Undefined expression type!",
                         form)
   fi
case = decl then
   if consp (form) ∧ top-prefix-p (form, nil)
    then case on statement-opr (form):
          case = fp-spec then
             {  pre := env;
                exc := coerce-to-subtype-exc (top-value (env),
                                              fp-spec-type (form),
                                              nil);
                ⟨nil, push-ese (make-constrained-entry (fp-spec-id (form),
                                                         constrain-range-if-necessary (
                                                                fp-spec-type (form),
                                                                top-value (env)),
                                                         coerce-to-subtype-val (
                                                             top-value (env),
                                                             fp-spec-type (form))),
                                 pop-value (env))⟩
                check-asserts (env_e, pre_e)
                 }
          case = assert1 then  check-assert (arg1 (form))
          case = invariant then
             {  check-assert (arg1 (form))
                ⟨nil, push-ase (arg1 (form), env)⟩
                 }
          case = object-decl then
             {  pre := env;
                I_e (object-decl-body (form), env, c)
                exc := coerce-to-subtype-exc (top-value (env),
                                              object-decl-type (form),
                                              nil);
                ⟨nil, push-ese (make-constrained-entry (object-decl-id (form),
                                                         constrain-range-if-necessary (
                                                                object-decl-type (form),
                                                                top-value (env)),
                                                         coerce-to-subtype-val (
                                                             top-value (env),
                                                             object-decl-type (form))),
                                 pop-value (env))⟩
                check-asserts (env_e, pre_e)
                 }
```

```
                    case = number-decl then
                      { pre := env;
                        I_e (number-decl-body (form), env, c)
                        ⟨nil, push-ese (make-entry (number-decl-id (form),
                                                    *base-integer*,
                                                    top-value (env)),
                                         pop-value (env))⟩
                        check-asserts (env_e, pre_e)
                        }
                    case = procedure then ⟨nil, extend-env-with-procedure (form,
                                                                            env)⟩
                    case = package then ⟨nil, extend-env-with-package (form,
                                                                        env)⟩
                    otherwise hard-error-env (env,
                                              "Undefined declaration type!",
                                              form)
                   endcase
            else hard-error-env (env,
                                 "Bad declaration (not prefix) : ",
                                 flg)
           fi
        otherwise hard-error-env (env,
                                  "No flag named : ",
                                  flg)
       endcase
 else hard-error-env (env, "Bad env ", form)
fi
Measure: interpret-measure (form, c)
```

The following supports the definition of the function interpret-program, which is used to actually interpret a main program in the context of a library.

The packages STANDARD and ADA bound in constant **annex-a** (see A.7).

CONSTANT:
some-real-big-integer = `1000000000000000000`

CONSTANT:
initial-env = `'((nil) (nil) (nil))`

DEFINITION:
```
find-package-decl (id, l)
 =
if atom (l)  then nil
 elseif id = arg1 (arg1 (car (l)))  then arg1 (car (l))
 else find-package-decl (id, cdr (l))
fi
```

DEFINITION:
```
merge-package (p, l)
 =
let decl be find-package-decl (arg1 (p), l)
    in
if ¬ decl  then mk-package (arg1 (p), nil, nil, arg2 (p), arg3 (p))
 else mk-package (arg1 (p),
                  arg2 (decl),
                  arg3 (decl),
                  arg2 (p),
                  arg3 (p))
```

```
fi

DEFINITION:
package-up (comp-units, decls)
 =
if null (comp-units)  then nil
 elseif ¬ comp-unit-p (car (comp-units))  then nil
 else let unit be car (comp-units)
         in
      case on car (arg1 (unit)):
        case = package-decl then  package-up (cdr (comp-units),
                                              cons (unit,
                                                    decls))
        case = package-body then
          cons (mk-comp-unit (merge-package (arg1 (unit),
                                             cdr (comp-units)
                                           @  decls),
                              arg2 (unit)),
                package-up (cdr (comp-units), decls))
        case = procedure then  cons (unit,
                                     package-up (cdr (comp-units),
                                                 decls))
        otherwise nil
       endcase
fi

DEFINITION:
interpret-program (main, library)
 =
if top-prefix-p (mk-compilation (library), nil)
 then ⟨exc, env⟩ := I×_d (package-up (library, nil) @ *annex-a*, *initial-env*, *some-real-big-integer*);

         if null (exc)
          then I_s (['proc-call, main, nil], env, *some-real-big-integer*)
          else hard-error-env (env, "Initial library elaboration failure")fi
 else hard-error-env (nil,"Library not a compilation")
fi
```

2.1 Subsidiary Routines

SET CURRENT PACKAGE to be **ACL2**.

INCLUDING the book: *macros*.

INCLUDING the book: *ilet*.

INCLUDING the book: *get-tree*.

INCLUDING the book: *insert-sort*.

INCLUDING the book: *subprefix-norm*.

INCLUDING the book: *type-check-macros*.

Entity stack

DEFINITION:
es-p (l)
 =

if ¬ consp (*l*) **then** null (*l*)
 else entry-p (car (*l*)) ∧ es-p (cdr (*l*))
fi

Value stack

DEFINITION:
vs-p (*l*)
 =
if ¬ consp (*l*) **then** null (*l*)
 else literal-p (car (*l*)) ∧ vs-p (cdr (*l*))
fi

Assertion stack

DEFINITION:
as-p (*l*)
 =
if ¬ consp (*l*) **then** null (*l*)
 else lexpr-p (car (*l*)) ∧ as-p (cdr (*l*))
fi

Stack of entity stack

DEFINITION:
es-p* (*l*)
 =
if ¬ consp (*l*) **then** null (*l*)
 else es-p (car (*l*)) ∧ es-p* (cdr (*l*))
fi

Stack of value stack

DEFINITION:
vs-p* (*l*)
 =
if ¬ consp (*l*) **then** null (*l*)
 else vs-p (car (*l*)) ∧ vs-p* (cdr (*l*))
fi

Stack of assertion stack

DEFINITION:
as-p* (*l*)
 =
if ¬ consp (*l*) **then** null (*l*)
 else as-p (car (*l*)) ∧ as-p* (cdr (*l*))
fi

DEFINITION:
env-p (*e*)
 =
true-listp (*e*) ∧ ((|*e*|) $=_n$ 3) ∧ es-p* (e_e) ∧ vs-p* (e_v) ∧ as-p* (e_a)

2.1.1 Constants

We would prefer to encapsulate 'ava-min-int' and 'ava-max-int' without providing a definition in order to reason about their effects generically. E.g

BEGIN ENCAPSULATION

CONSTRAIN the functions:

FUNCTION: ava-min-int

ACCORDING TO THE FOLLOWING EVENTS:

LOCAL DEFINITION:
ava-min-int = -32000

END ENCAPSULATATION.

But if we do so, the interpreter is not executable.

So, in order to execute functions that depend on the integer bounds, we use the following, which gives 'ava-min-int' and 'ava-max-int' fixed values.

DEFINITION:
ava-min-int = -32000

DEFINITION:
ava-max-int = 32000

THEOREM: integerp-ava-min-int *:type-prescription*
integerp (ava-min-int) ∧ (ava-min-int < 0)

THEOREM: integerp-ava-max-int *:type-prescription*
integerp (ava-max-int) ∧ (0 < ava-max-int)

THEOREM: min-int-close-to-max-int
(ava-max-int + ava-min-int) ≤ 1

MODIFY the current theory: Disable 'ava-min-int', 'ava-max-int', and their executable counterparts

TESTS and EXTRACTION

Basic extraction functions on prefix forms return the operator name, opr (*form*), a list of the arguments, arg* (*form*), and specific arguments, arg1 (*form*)..argn (*form*). Others extraction functions are defined according to the abstract syntax defined in "subprefix-norm.input". That file is used to generate the functions in "subprefix-norm.lisp", which are defined in .

2.1.2 Errors

Hard errors are errors in the semantics and always percolate all the way to the top. If we do everything right (and only pass in legal statically-checked programs), then we should never see a hard error. Someday we may want to prove that there is no hard error when interpreting a well-formed AVA program.

The simple AVA predefined exceptions are simply constants.

CONSTANT:

program-error = ’`program-error`

CONSTANT:
constraint-error = ’`constraint-error`

CONSTANT:
storage-error = ’`storage-error`

Defined only in AVA for purposes of tracking assertions:

CONSTANT:
logical-error = ’`logical-error`

The following are used to control the flow of execution.

CONSTANT:
loop-exit = ’`loop-exit`

CONSTANT:
subprogram-return = ’`subprogram-return`

MACRO:
logical-error (&REST *args*)
=
if *args*
then ‘`(list *logical-error*`
`,(car args)`
`(list ,@(cdr args)))`
else ‘`(list *logical-error*`
`"Failed annotation"``)`
fi

Added *logical expressions*, *lexprs*, which are just expressions in the ACL2 logic.

instate (*lexpr*)
outstate (*lexpr*)

qinstate and ‘outstate’ are macros that expand into

let *env* **be** *x*
in
eval (*lexpr*)

, where *x* is the input or output state.

AVA predefined exception, constraint_error

Note that constraint-error exceptions have a little more structure than the others. This enables us to pass some debugging information out with what would otherwise be an uninformative exception. The semantics does not distinguish these otherwise.

DEFINITION:
handler-error-p (*exc*)
=
(listp (*exc*) $\wedge$ (car (*exc*) = **constraint-error**))
$\vee$ (*exc* = **program-error**)
$\vee$ (*exc* = **storage-error**)

MACRO:

constraint-error (*fmt-string*, &REST *args*)
=
```
‘(list *constraint-error*
  ,fmt-string
  (list ,@args))
```

DEFINITION:
extend-constraint-error (*exc*, *new-args*)
=
if true-listp (*exc*)
$\wedge$ (car (*exc*) $=_{eq}$ **constraint-error**)
$\wedge$ stringp (cadr (*exc*)) **then** [car (*exc*), cadr (*exc*), caddr (*exc*) @ *new-args*]
else *exc*
fi

2.1.3 Ava Literals, Values and Types

A literal is either an integer, a character, a string, an array literal, or a record literal.

2.1.4 Operators

The elements of *ava-operators-alist* are the predefined operators and their arity.

CONSTANT:
ava-operators-alist =
```
'((unary-plus . 1)
 (plus . 2)
 (minus . 2)
 (multiply . 2)
 (divide . 2)
 (mod . 2)
 (rem . 2)
 (abs . 1)
 (power . 2)
 (equal . 2)
 (ne . 2)
 (in . 2)
 (in-range . 2)
 (in-type . 2)
 (lt . 2)
 (array-< . 2)
 (not . 1)
 (and . 2)
 (or . 2)
 (xor . 2)
 (catenate . 2)
 (array-not . 1)
 (array-and . 2)
 (array-or . 2)
 (if . 3)
)
```

DEFINITION:
ava-operators-alist = **ava-operators-alist**

Let’s state all the facts we’ll need about this alist, and then disable it. This is an example of the utility of macros in ACL2.

DEFINITION:

ava-op-defthm-forms (*op-alist*)
 =
if *op-alist*
 then cons (‘(defthm
```
            ,(pack2
              'ava-op-
              (caar op-alist))
            (equal
             (assoc-eq
              ',(caar op-alist)
              (ava-operators-alist))
             ',(car op-alist))),
```
          ava-op-defthm-forms (cdr (*op-alist*)))
 else nil
fi

MACRO:
prove-ava-op-defthms
 =
cons (’progn, ava-op-defthm-forms (**ava-operators-alist**))

EVENT:
```
(prove-ava-op-defthms)
```

THEOREM: symbol-alistp-ava-operators-alist *:type-prescription*
symbol-alistp (ava-operators-alist) = **t**

MODIFY the current theory:

Disable ‘ava-operators-alist’ and the executable counterpart of ‘(ava-operators-alist)’.

2.1.5 Expression evaluation

This is now handled with statement evaluation, by Interpret, which returns two values, an exception (or NIL) and an environment. The result of the evaluation is top-value (*env*), which is top (top (env_v)). For example,

I_e (3 + 4, *env*) => ⟨**nil**, push-value (7, *env*)⟩

At one point we checked that all variables are bound, in order to say that we have an expression. But this requirement turned into an analogous requirement for statements, which in turn forced proof obligations that if the handler of a block is a statement-p with respect to a given variable stack, then it’s still one even after we interpret the body of that block. Since we probably want to support the notion of unbound anyhow, we just allow variables to be unbound and check things dynamically.

The operator functions defined below all return a first value of nil (normal) unless the arguments require an exception to be raised.

DEFINITION:
fix-int (*x*)
 =
if integerp (*x*) **then** *x*
 else 0
fi

DEFINITION:
fix-bool (*x*)
 =

if x **then** `'(true)`
else `'(false)`
fi

DEFINITION:
int-not-in-range $(val, lower, upper)$
$=$
$(val < lower) \vee (upper < val)$
Guard: rationalp $(val) \wedge$ rationalp $(lower) \wedge$ rationalp $(upper)$

Let's define the AVA operators and then disable them all at once.

2.1.6 Numeric Operators

Notice that we don't neet to allow different base types of integer.

LABEL: ava-op-fns-start

DEFINITION:
ava-plus-exc (x, y)
$=$
if int-not-in-machine-range $(x + y)$
then constraint-error (**`"Integer overflow, (+ ~p0 ~p1)."`**, x ,y)
else nil
fi

DEFINITION:
ava-plus-val $(x, y) = x + y$

DEFINITION:
ava-multiply-exc (x, y)
$=$
if int-not-in-machine-range $(x \times y)$
then constraint-error (**`"Integer overflow, (* ~p0 ~p1)."`**, x, y)
else nil
fi

DEFINITION:
ava-multiply-val $(x, y) = x \times y$

DEFINITION:
ava-power-exc (x, y)
$=$
if int-not-in-machine-range (expt (x, y))
then constraint-error (**`"Integer overflow, (* ~p0 ~p1)."`**, x, y)
elseif $y < 0$
then constraint-error (**`"Exponent underflow, (* ~p0 ~p1)."`**, x, y)
else nil
fi

DEFINITION:
ava-power-val $(x, y) =$ expt (x, y)

Relations between division, mod and rem.

a = (a/b)*b + (a rem b)
(a rem b) has sign of a and abs(a rem b) < abs(b)
(-a)/b = -(a/b) = a/(-b)

It should be a theorem that: $a = (($truncate $(a, b) \times b) +$ rem $(a, b))$

DEFINITION:
ava-divide-exc (x, y)
 =
if $y = 0$
 then constraint-error (`"Division by 0, (/ ~p0 ~p1)."`, x, y)
 elseif int-not-in-machine-range (truncate (x, y))
 then constraint-error (`"Integer overflow, (/ ~p0 ~p1)."`, x, y)
 else nil
fi

DEFINITION:
ava-divide-val (x, y) = truncate (x, y)

The possibility of the overflow exception is due to the case (*ava_min_int* rem -1).

DEFINITION:
ava-rem-exc (x, y)
 =
if $y = 0$
 then constraint-error (`"Zero divisor, ~p0 rem ~p1."`, x, y)
 elseif int-not-in-machine-range (rem (x, y))
 then constraint-error (`"Integer overflow, (~p0 rem ~p1)."`, x, y)
 else nil
fi

DEFINITION:
ava-rem-val (x, y) = rem (x, y)

DEFINITION:
ava-mod-exc (x, y)
 =
if $y = 0$
 then constraint-error (`"Zero divisor, ~p0 mod ~p1."`, x, y)
 elseif int-not-in-machine-range (mod (x, y))
 then constraint-error (`"Integer overflow, (~p0 mod ~p1)."`, x, y)
 else nil
fi

DEFINITION:
ava-mod-val (x, y) = mod (x, y)

DEFINITION:
ava-unary-minus-exc (x)
 =
if int-not-in-machine-range ($- x$)
 then constraint-error (`"Integer overflow, (- ~p0)."`, x)
 else nil
fi

DEFINITION:
ava-unary-minus-val (x) = $- x$

2.1.7 Boolean Operators

Important note: Booleans literals in ACL2 are **t** and **nil** (or non-**t**). Booleans literals in AVA are ‘true’ or ‘false’. This somewhat unfortunate circumstance was necessitated by the need to distinguish trees from leaves in array and record literals. E.g. we needed the following theorems:

THEOREM: tree-not-leaf
treep (x) $\rightarrow$ ($\neg$ leafp (x))

THEOREM: leaf-not-tree
leafp (x) → (¬ treep (x))

Regarding equality:

Array and record values are alists of the form: ((index . value)*). Two such values are equal if corresponding elements of their values are. We use a single equality function rather than generating the numerous type specific versions required by naive adherence to the manual. This makes it much more tractable to provide a library of predefined lemmas for reasoning about equality.

DEFINITION:
minimum (l)
=
if ¬ consp (l) **then** `0`
elseif ¬ rationalp (car (l)) **then** minimum (cdr (l))
elseif ¬ consp (cdr (l)) **then** car (l)
else min (car (l), minimum (cdr (l)))
fi

DEFINITION:
maximum (l)
=
if ¬ consp (l) **then** `-1`
elseif ¬ rationalp (car (l)) **then** maximum (cdr (l))
elseif ¬ consp (cdr (l)) **then** car (l)
else max (car (l), maximum (cdr (l)))
fi

DEFINITION:
array-literal-from (*literal*) = minimum (range (*literal*))

DEFINITION:
array-literal-to (*literal*) = maximum (range (*literal*))

Note that the null record and array case are handled, since if x and y aren't equal, they cannot both be non-empty.

DEFINITION:
ava-equal-val (x, y)
=
if $x = y$ **then** true
elseif (¬ consp (x)) ∨ (¬ consp (y)) **then** false
elseif array-literal-p (x) ∧ array-literal-p (y)
then fix-bool (set-equal (x, y))
elseif record-literal-p (x) ∧ record-literal-p (y)
then fix-bool (set-equal (x, y))
else false
fi

THEOREM: char-code-nonnegative-integerp-for-ada-char-p *:type-prescription*
ada-char-p (x) → (integerp (char-code (x)) ∧ (`0` ≤ char-code (x)))

MODIFY the current theory:

Disable 'standard-char-p', 'char-code' and 'ada-char-p'.

DEFINITION:
$x <_n y$
=
if rationalp (x)
then if rationalp (y) **then** $x < y$

else nil
fi
else nil
fi

DEFINITION:
$x >_n y = \neg (x <_n y)$

DEFINITION:
$x \leq_n y = (x <_n y) \vee (x = y)$

DEFINITION:
$x \geq_n y = (x >_n y) \vee (x = y)$

THEOREM: le-1
$(x <_n y) \rightarrow (\neg (y \leq_n x))$

THEOREM: le-2
$(y \leq_n x) \rightarrow (\neg (x <_n y))$

THEOREM: le-3
$x \leq_n x$

THEOREM: ge-le-eq
$((x \leq_n y) \wedge (x \geq_n y)) \rightarrow (x = y)$

DEFINITION:
ava-<-val (*x*, *y*)
=
fix-bool (**if** ada-char-p (*x*) $\wedge$ ada-char-p (*y*)
then char-code (*x*) < char-code (*y*)
elseif rationalp (*x*) $\wedge$ rationalp (*y*) **then** $x <_n y$
else nil
fi)

Treat non-rational lower and upper as neg infinity and pos infinity, respectively

DEFINITION:
between (*val*, *lower*, *upper*)
=
if $\neg$ rationalp (*val*) **then nil**
else $(lower \leq_n val) \wedge (val \leq_n upper)$
fi

DEFINITION:
ava-in-range-val (*val*, *lower*, *upper*)
=
fix-bool (**if** ada-char-p (*val*) $\wedge$ ada-char-p (*lower*) $\wedge$ ada-char-p (*upper*)
then between (char-code (*val*), char-code (*lower*), char-code (*upper*))
else between (*val*, *lower*, *upper*)
fi)

2.1.8 Unary Numeric Operators

DEFINITION:
ava-abs-exc (*x*)
=
let *y* **be** |ifix (*x*)|
in
if int-not-in-machine-range (*y*)
then constraint-error (`"Integer overflow, (abs ~p0)."`, *x*)

else nil
fi

DEFINITION:
ava-abs-val (*x*) =|ifix (*x*)|

DEFINITION:
ava-not (*x*)
=
if true-p (*x*) **then** false
else true
fi

DEFINITION:
ava-and (*x*, *y*)
=
if true-p (*x*) **then** *y*
else false
fi

DEFINITION:
ava-or (*x*, *y*)
=
if true-p (*x*) **then** true
else *y*
fi

DEFINITION:
ava-xor (*x*, *y*)
=
if true-p (*x*)
then if false-p (*y*) **then** true
else false
fi
elseif true-p (*y*) **then** true
else false
fi

DEFINE the theory **ava-op-fns** to be

the current function theory *less* the function theory **ava-op-fns-start**.

MODIFY the current theory:

Disable 'ava-op-fns'.

2.1.9 Application of Operators to Evaled Arguments

This function defines the behavior of the built-in functions. It is used in 'interpret-eval'. Note that the arguments have already been evaluated without an exception. This evaluation is of course done in 'interpret-eval'.

DEFINITION:
eval-opr-val (*opr*, *args*)
=
case on *opr*:
case = `plus` **then** ava-plus-val ($args_0$, $args_1$)
case = `unary-minus` **then** ava-unary-minus-val ($args_0$)
case = `minus` **then** ava-plus-val ($args_0$, $-\ args_1$)
case = `multiply` **then** ava-multiply-val ($args_0$, $args_1$)

case = `divide` **then** ava-divide-val ($args_0$, $args_1$)
case = `mod` **then** ava-mod-val ($args_0$, $args_1$)
case = `rem` **then** ava-rem-val ($args_0$, $args_1$)
case = `abs` **then** ava-abs-val ($args_0$)
case = `power` **then** ava-power-val ($args_0$, $args_1$)
case = `equal` **then** ava-equal-val ($args_0$, $args_1$)
case = `ne` **then** ava-not (ava-equal-val ($args_0$, $args_1$))
case = `in-range` **then** ava-in-range-val ($args_0$, $args_1$, $args_2$)
case = `lt` **then** ava-<-val ($args_0$, $args_1$)
case = `gt` **then** ava-<-val ($args_1$, $args_0$)
case = `le` **then**
  if false-p (ava-<-val ($args_0$, $args_1$))
   then ava-equal-val ($args_0$, $args_1$)
   else true
  fi
case = `ge` **then**
  if false-p (ava-<-val ($args_1$, $args_0$))
   then ava-equal-val ($args_0$, $args_1$)
   else true
  fi
case = `not` **then** ava-not ($args_0$)
case = `and` **then** ava-and ($args_0$, $args_1$)
case = `or` **then** ava-or ($args_0$, $args_1$)
case = `xor` **then** ava-xor ($args_0$, $args_1$)
otherwise nil
endcase

VERIFY GUARDS for ‘top-prefix-p’

DEFINITION:
eval-opr-exc (*opr*, *args*)
=
case on *opr*:
case = `plus` **then** ava-plus-exc ($args_0$, $args_1$)
case = `unary-minus` **then** ava-unary-minus-exc ($args_0$)
case = `minus` **then** ava-plus-exc ($args_0$, $-$ $args_1$)
case = `multiply` **then** ava-multiply-exc ($args_0$, $args_1$)
case = `divide` **then** ava-divide-exc ($args_0$, $args_1$)
case = `mod` **then** ava-mod-exc ($args_0$, $args_1$)
case = `rem` **then** ava-rem-exc ($args_0$, $args_1$)
case = `abs` **then** ava-abs-exc ($args_0$)
case = `power` **then** ava-power-exc ($args_0$, $args_1$)
otherwise nil
endcase

MODIFY the current theory:

Disable ‘eval-opr-val’ and ‘eval-opr-exc’.

2.2 Arrays and Records

If the constraint isn't 'range-p', then it's 'unconstrained-p'.

DEFINITION:
satisfies-range-constraint (*n*, *constraint*)
 =
if range-p (*constraint*)
 then between (*n*, range-from (*constraint*), range-to (*constraint*))
 else unconstrained-p (*constraint*)
fi

Let's save a lot of case splits.

MODIFY the current theory:

Disable 'nth'.

DEFINITION:
range-size (*from*, *to*)
 =
if $from \le_n to$ **then** 1 + (*to* − *from*)
 else 0
fi

2.3 Types, including Conversion and Qualification

For the coercion functions below:

literal = empty | boolean-literal | numeric-literal | ada-char | array-literal | record-literal
type = record-type | array-type | predefined-type | range

We assume when this is called that the base type of the literal equals the base type of the type. By "base type" we mean integer for range types, the corresponding unconstrained array type for array types, and the type itself otherwise.

DEFINITION:
adjust-array-literal (*literal*, *delta*)
 =
if ¬ consp (*literal*) **then** *literal*
 elseif ¬ (consp (car (*literal*)) ∧ integerp (caar (*literal*)))
 then *literal*
 else cons (cons (caar (*literal*) + *delta*, cdar (*literal*)),
 adjust-array-literal (cdr (*literal*), *delta*))
fi

DEFINITION:
constrain-array-val (*literal*, *from*)
 =
let *old* **be** array-literal-from (*literal*)
 in
if *old* = *from* **then** *literal*
 else adjust-array-literal (*literal*, *from* − *old*)
fi

If ¬ *strong-flg* just compare length, otherwise also check the equality of lower bounds.

DEFINITION:
constrain-array-exc (*literal*, *from*, *to*, *strong-flg*)

```
 =
if (|literal|) = range-size (from, to)
 then if strong-flg
         then if from = array-literal-from (literal)  then nil
                 else constraint-error ("Literal ~p0 does not have base of ~p1.", literal)
                fi
         else nil
        fi
 else constraint-error ("Length of literal ~p0 is not equal to ~p1.",
                                  literal,
                                  range-size (from, to))
fi
```

DEFINITION:
coerce-to-subtype-from-lit-val (*literal*, *old-lit*)

```
 =
if array-literal-p (literal)
 then constrain-array-val (literal, array-literal-from (old-lit))
 else literal
fi
```

DEFINITION:
coerce-to-subtype-from-lit-exc (*literal*, *old-lit*, *typ*)

```
 =
if array-literal-p (literal)
 then constrain-array-exc (literal,
                                       array-literal-from (old-lit),
                                       array-literal-to (old-lit),
                                       nil)
 elseif range-p (typ)
 then if satisfies-range-constraint (literal, typ)  then nil
         else constraint-error ("Literal ~p0 is out of range for type ~p1.", literal, typ)
        fi
 else nil
fi
```

DEFINITION:
coerce-to-subtype-val (*literal*, *typ*)

```
 =
if array-type-p (typ)
 then let constraint be array-type-index (typ)
            in
        if range-p (constraint)
         then constrain-array-val (literal, range-from (constraint))
         else literal
        fi
 else literal
fi
```

DEFINITION:
coerce-to-subtype-exc (*literal*, *typ*, *strong-flg*)

```
 =
if array-type-p (typ)
 then let constraint be array-type-index (typ)
            in
        if range-p (constraint)
         then constrain-array-exc (literal,
                                               range-from (constraint),
                                               range-to (constraint),
                                               strong-flg)
```

```
         else nil
        fi
 elseif range-p (typ)
 then if satisfies-range-constraint (literal, typ)  then nil
         else constraint-error ("Literal ~p0 is out of range for type ~p1.", literal, typ)
        fi
 else nil
fi
```

Type-convert converts an expression to a base type, then checks that it satisfies subtype requirements.

Conversion checks:

- Numeric
  1. Base check is NOOP. Check range.
- Array
  1. Same dimensions.
  2. Index types the same or convertible.
  3. Component types the same.
  4. Constraints on component types the same.
  5. If type is unconstrained then bounds come from converting indices to base type of unconstrained index type. (Which is a noop, since the base type must be INTEGER.)

Raise constraint-error if numeric conversions fail to satisfy constraint.

DEFINITION:

```
type-convert-val (expr, typ)
 =
if      array-type-p (typ)
   ∧   array-literal-p (expr)
   ∧   range-p (array-type-index (typ))
 then constrain-array-val (expr, range-from (array-type-index (typ)))
 else expr
fi
```

Note that expr cannot be an aggregate or a string literal.

DEFINITION:

```
type-convert-exc (expr, typ)
 =
if array-type-p (typ)
 then let constraint be array-type-index (typ)
           in
        if array-literal-p (expr) ∧ range-p (constraint)
         then if       (|expr|)
                   =   range-size (range-from (constraint),
                                    range-to (constraint))  then nil
                else constraint-error (
                                  "Convert: Range of literal ~p0 doesn't satisfy constraint ~p1.",
                                  expr,
                                  constraint)
               fi
         else nil
        fi
 elseif range-p (typ)
```

then if satisfies-range-constraint (*expr*, *typ*) **then nil**
    else constraint-error (`"Convert: Value ~p0 does not satisfy subtype range ~p1."`,
expr,
typ)
fi
else nil
fi

2.4 More Expression Evaluation

array-literal := [[*i* . *val]* ...]
record-literal := [[**f** . *val]* ...]

array-literal-p (*form*) == treep (*form*) ∧ all-labels-integer (domain (*form*))
record-literal-p (*form*) == treep (*form*) ∧ all-labels-id (domain (*form*))

DEFINITION:
get-array-elem-exc (*array-literal*, *int*)
=
let *from* **be** array-literal-from (*array-literal*),
to **be** array-literal-to (*array-literal*)
in
if between (*int*, *from*, *to*) **then nil**
else constraint-error (`"Array index out of bounds, index ~p0."`,
int)
fi

Note that this does not logically guarantee that an element is present in an array-literal. Our predicates for arrays do not guarantee that all values between array-literal-from (*a*) and array-literal-to (*a*) are present in the alist that represents *a*. At the moment this is an implicit assumption that may need to be made explicit.

THEOREM: get-array-elem-exc-null-array
((¬ consp (*x*)) ∧ integerp (*i*)) → get-array-elem-exc (*x*, *i*)

AXIOM: array-elem-exists
(array-literal-p (*x*)
∧ integerp (*i*)
∧ between (*i*,
array-literal-from (*array-literal*),
array-literal-to (*array-literal*)))
→ exists-t (*x*, *i*)

2.5 Statement Evaluation Support

DEFINITION:
variable-update-1 (*var*, *val*, *stack*)
=
if consp (*stack*)
then let *entry* **be** car (*stack*),
rest **be** cdr (*stack*)
in
if basic-entry-p (*entry*) ∧ (entry-name (*entry*) = *var*)
then cons (make-entry (*var*, entry-decl (*entry*), *val*), *rest*)
else cons (*entry*, variable-update-1 (*var*, *val*, *rest*))
fi

 else nil
fi

var is an ‘id-p’

DEFINITION:
variable-update (*var*, *val*, *var-stacks*)
 =
if consp (*var-stacks*)
 then if assoc-equal (*var*, car (*var-stacks*))
  then cons (variable-update-1 (*var*, *val*, car (*var-stacks*)),
   cdr (*var-stacks*))
  else cons (car (*var-stacks*),
   variable-update (*var*, *val*, cdr (*var-stacks*)))
 fi
 else *var-stacks*
fi

2.5.1 Assignment

We are about to define the functions that break apart variable references, e.g. v[i].j[k] => ⟨*v*, [*iv*, *j*, *kv*]⟩ where *iv* and *kv* are the values obtained by evaluating *i* and *k*, respectively.

THEOREM: indexed-component-root-decrease *:linear*
 (consp (cdr (*x*)) ∧ indexed-component-p (*x*))
→ (acl2-count (indexed-component-root (*x*)) < acl2-count (*x*))

THEOREM: selected-component-root-decrease *:linear*
 (consp (cdr (*x*)) ∧ selected-component-p (*x*))
→ (acl2-count (selected-component-root (*x*)) < acl2-count (*x*))

MODIFY the current theory: Disable ‘selected-component-root’, ‘selected-component-p’, ‘indexed-component-root’ and ‘indexed-component-p’.

DEFINITION:
root* (*x*)
 =
if ¬ (consp (*x*) ∧ consp (cdr (*x*))) **then** *x*
 elseif indexed-component-p (*x*) **then** root* (indexed-component-root (*x*))
 elseif selected-component-p (*x*) **then** root* (selected-component-root (*x*))
 else *x*
fi

we assume it’s an id-p

DEFINITION:
actions (*x*)
 =
if ¬ (consp (*x*) ∧ consp (cdr (*x*))) **then nil**
 elseif indexed-component-p (*x*)
 then cons (indexed-component-index (*x*), actions (indexed-component-root (*x*)))
 elseif selected-component-p (*x*)
 then cons (selected-component-field (*x*),
  actions (selected-component-root (*x*)))
 else nil
fi

An exception will be raised if the index is not in range. To simplify, while it should not happen, an exception will also be raised if the variable is not in the value stack.

Returns the updated value. Handles implicit array conversion. We don't use 'set-l' beause of the coercion that may occur at the leaf. Assumes that the actions are appropriate for *root-val*.

DEFINITION:
```
update-value-val (actions, root-val, splice-val)
 =
if ¬ consp (actions)
 then coerce-to-subtype-from-lit-val (splice-val, root-val)
 else put-t (root-val,
             car (actions),
             update-value-val (cdr (actions),
                               get-t (root-val, car (actions)),
                               splice-val))
fi
```

DEFINITION:
```
update-value-exc (root-type, actions, root-val, splice-val)
 =
if ¬ consp (actions)
 then coerce-to-subtype-from-lit-exc (splice-val, root-val, root-type)
 elseif array-literal-p (root-val)
 then if satisfies-range-constraint (car (actions),
                                     array-type-index (root-type))
       then   get-array-elem-exc (root-val, car (actions))
            ∨ update-value-exc (array-type-elements (root-type),
                                cdr (actions),
                                get-t (root-val, car (actions)),
                                splice-val)
       else constraint-error (
                 "Component index out of bounds, index ~p0, actions ~p1, in ~p2.",
                 car (actions),
                 cdr (actions))
      fi
 else update-value-exc (field-spec-decl (arg* (root-type)_car (actions)),
                        cdr (actions),
                        get-t (root-val, car (actions)),
                        splice-val)
fi
```

A "variable" may have unevaluated indices; once we've evaluated the indices, it's still a variable but we allow ourselves to call it a "reference".

DEFINITION:
```
assign-to-env (id, actions, splice-val, env)
 =
let* entry be variable-lookup (id, env),
     root-val be entry-value (entry),
     root-type be entry-decl (entry),
     exc be update-value-exc (root-type, actions, root-val, splice-val)
     in
if exc then ⟨extend-constraint-error (exc, [id]), env⟩
 else ⟨nil, set-es (variable-update (id,
                                     update-value-val (actions,
                                                       root-val,
                                                       splice-val),
                                     env_e),
                    env)⟩
fi
```

2.5.2 Procedure Call Support

DEFINITION:
constrain-range-if-necessary (*typ*, *literal*)
 =
if array-type-p (*typ*) ∧ unconstrained-p (array-type-index (*typ*))
 then mk-array-type (mk-range (array-literal-from (*literal*),
                                              array-literal-to (*literal*)),
                              array-type-elements (*typ*))
 else *typ*
fi

DEFINITION:
make-constrained-entry (*id*, *typ*, *val*)
 =
make-entry (*id*, constrain-range-if-necessary (*typ*, *val*), *val*)

THEOREM: len-0
((|*x*|) = `0`) = atom (*x*)

2.6 Interpreter

In our statement interpreter, the clock is decremented just before subprogram calls and while-loop recursions. A clock of 0 means "out of time".

CONSTANT:
out-of-time-msg = **`"Out of time!"`**

MACRO:
pop-variable-stack (*n*, *env*)
 =
`‘(nthcdr ,n ,env)`

MACRO:
interpret-stmt (*stmt*, *env*, *clock*)
 =
[`’interpret-eval`, `’’stmt`, *stmt*, *env*, *clock*]

MACRO:
interpret-stmts (*stmt-list*, *env*, *clock*)
 =
[`’interpret-eval`, `’’stmts`, *stmt-list*, *env*, *clock*]

MACRO:
interpret-exp (*exp*, *env*, *clock*)
 =
[`’interpret-eval`, `’’exp`, *exp*, *env*, *clock*]

MACRO:
interpret-exps (*exp-list*, *env*, *clock*)
 =
[`’interpret-eval`, `’’exps`, *exp-list*, *env*, *clock*]

MACRO:
interpret-decl (*decl*, *env*, *clock*)
 =
[`’interpret-eval`, `’’decl`, *decl*, *env*, *clock*]

MACRO:
interpret-decls (*decl-list*, *env*, *clock*)

=
[`'interpret-eval`, `''decls`, *decl-list*, *env*, *clock*]

DEFINITION:
statement-opr (*stmt*) = car (*stmt*)

THEOREM: statement-opr-non-nil-implies-consp-stmt *:forward-chaining*
statement-opr (*stmt*) $\rightarrow$ consp (*stmt*)

DEFINITION:
top-prefix-decls-p2 (*l*)
=
if ¬ consp (*l*) **then t**
elseif decl-p (car (*l*)) **then** top-prefix-decls-p2 (cdr (*l*))
else nil
fi

DEFINITION:
top-prefix-decls-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif top-prefix-decls-p2 (car (*l*)) **then** top-prefix-decls-p (cdr (*l*))
else nil
fi

DEFINITION:
conjunction (*l*)
=
if null (*l*) **then** true
elseif ¬ consp (*l*) **then** *l*
else cons (`'and`, *l*)
fi

DEFINITION:
extract-agg-values (*l*)
=
if atom (*l*) **then nil**
elseif aggregate-pos-p (car (*l*))
then cons (aggregate-pos-value (car (*l*)), extract-agg-values (cdr (*l*)))
elseif aggregate-choice-p (car (*l*))
then cons (aggregate-choice-value (car (*l*)), extract-agg-values (cdr (*l*)))
else nil
fi

DEFINITION:
extract-agg-labels1 (*l*, *i*)
=
if atom (*l*) **then nil**
elseif aggregate-pos-p (car (*l*))
then cons (*i*, extract-agg-labels1 (cdr (*l*), *i* + `1`))
elseif aggregate-choice-p (car (*l*))
then let *choice* **be** aggregate-choice-choice (car (*l*))
in
if others-p (*choice*) **then** [`'others`]
elseif id-p (*choice*)
then cons (id-root (*choice*),
extract-agg-labels1 (cdr (*l*), *i* + `1`))
else nil
fi
else nil
fi

DEFINITION:
extract-agg-labels (l) = extract-agg-labels1 (l, 0)

DEFINITION:
all-agg-pos (l)
=
if atom (l) **then t**
else aggregate-pos-p (car (l)) $\land$ all-agg-pos (cdr (l))
fi

DEFINITION:
all-agg-choice (l)
=
if atom (l) **then t**
else aggregate-choice-p (car (l)) $\land$ all-agg-choice (cdr (l))
fi

DEFINITION:
pos-aggregate-p (x)
=
aggregate-p (x) $\land$ all-agg-pos (arg* (x))

DEFINITION:
choice-aggregate-p (x)
=
aggregate-p (x) $\land$ all-agg-choice (arg* (x))

Remember the form of ‘array-type-p’:

array-type == **array-type** index : *type-mark*, elements : *type*
type-mark == **type-mark** type : *id*, constraint : *constraint*
constraint == *subtype* | *unconstrained* | *range* | *attribute*

DEFINITION:
create-inclusive-list (*from*, *to*)
=
if ($\neg$ integerp (*from*)) $\lor$ ($\neg$ integerp (*to*)) **then nil**
elseif *from* > *to* **then nil**
elseif *from* = *to* **then** [*to*]
else cons (*from*, create-inclusive-list (*from* + 1, *to*))
fi
Measure: ifix (max (0, *to* − *from*))

MODIFY the current theory:

Enable ‘nth’.

DEFINITION:
compute-type-range-list (x)
=
if type-mark-p (x) **then** compute-type-range-list (type-mark-constraint (x))
elseif id-p (x) **then** create-inclusive-list (ava-min-int, ava-max-int)
elseif range-p (x) **then** create-inclusive-list (range-from (x), range-to (x))
else nil
fi

DEFINITION:
field-spec-ids (*fields*)
=
if atom (*fields*) **then nil**
else cons (field-spec-id (car (*fields*)), field-spec-ids (cdr (*fields*)))

fi

DEFINITION:
type-tree-labels (*typ*, *args*)
=
if (¬ atom (*args*)) ∧ symbolp (car (*args*)) **then** *args*
elseif record-type-p (*typ*) **then** field-spec-ids (arg* (*typ*))
elseif array-type-p (*typ*)
then compute-type-range-list (array-type-index (*typ*))
else nil
fi

DEFINITION:
extract-ids (*formals*)
=
if atom (*formals*) **then nil**
else cons (fp-spec-id (car (*formals*)), extract-ids (cdr (*formals*)))
fi

pre and post are environments.

DEFINITION:
assign-actuals (*formals*, *actuals*, *values*)
=
if atom (*formals*) **then nil**
elseif fp-spec-p (car (*formals*))
∧ variable-p (fp-spec-mode (car (*formals*)))
then cons (mk-assign (car (*actuals*), car (*values*)),
assign-actuals (cdr (*formals*), cdr (*actuals*), cdr (*values*)))
else assign-actuals (cdr (*formals*), cdr (*actuals*), cdr (*values*))
fi

DEFINITION:
extend-env-with-procedure (*form*, *env*)
=
push-ese (make-entry (procedure-id (*form*), *form*, **nil**), *env*)

DEFINITION:
extend-env-with-package (*form*, *env*)
=
push-ese (make-entry (package-id (*form*), *form*, **nil**), *env*)

Measure information for interpret-eval.

DEFINITION:
interpret-measure (*stmt*, *clock*)
=
cons (1 + **if** pos-int-p (*clock*) **then** *clock*
else 0
fi,
if *stmt* **then** acl2-count (*stmt*)
else 0
fi)

THEOREM: interpret-measure-facts
(pos-int-p (*x*)
→ (interpret-measure (*stmt2*, -1 + *x*) $<_{\varepsilon}$ interpret-measure (*stmt1*, *x*)))
∧ (*stmt2* → ((interpret-measure (*stmt1*, *c*) $<_{\varepsilon}$ interpret-measure (*stmt2*, *c*))
= (acl2-count (*stmt1*) < acl2-count (*stmt2*))))
∧ e0-ordinalp (interpret-measure (*stmt*, *n*))

MODIFY the current theory:

Enable ‘e0-ord-<’.

MODIFY the current theory:

Disable ‘interpret-measure’.

MODIFY the current theory:

Disable ‘top-prefix-p’.

THEOREM: count-actions *:linear*
$\text{consp}(\textit{form}) \rightarrow (\text{acl2-count}(\text{actions}(\textit{form})) < \text{acl2-count}(\textit{form}))$

THEOREM: count-actions-cadr *:linear*
$\text{consp}(\text{cadr}(\textit{form}))$
$\rightarrow (\text{acl2-count}(\text{actions}(\text{cadr}(\textit{form}))) < \text{acl2-count}(\text{cadr}(\textit{form})))$

THEOREM: count-actions-cadr-2 *:linear*
$(\text{consp}(\textit{form}) \wedge \text{consp}(\text{cadr}(\textit{form})))$
$\rightarrow (\text{acl2-count}(\text{actions}(\text{cadr}(\textit{form}))) < \text{acl2-count}(\textit{form}))$

THEOREM: acl2-count-extract-agg-values *:linear*
$\text{acl2-count}(\text{extract-agg-values}(w)) \leq \text{acl2-count}(w)$

2.7 Input

User input, from parsed AVA files, comes in as follows:

ACL2 constants contain the declarations and bodies of compilation_units. User defined ACL2 functions, axioms and theorems, present in the annotated AVA, have been literally extracted.

CONSTANT:
pattern_scope-decl = ...

Extracted ACL2

DEFINITION:
$\text{lowerp}(x) = (\text{'a'} \leq x) \wedge (x \leq \text{'z'})$

THEOREM: upperp-not-lowerp
$\text{upperp}(x) \rightarrow (\neg\, \text{lowerp}(x))$
...

CONSTANT:
**pattern_scope-body* = pending*

The Library is defined.

CONSTANT:
**library* = pending*

The main program is named.

CONSTANT:
main = `'(id main 1)`

Finally, a theorem is presented that states that there exists a permitted order of elaboration supporting the evaluation of *main*.

THEOREM: order-of-elaboration
some-order-exists(**main**, **library**)

Appendix A
Support

We depend on a number of ACL2 libraries that are part of the ACL2 V1.8 distribution. In particular:

public/sets
arithmetic/equalities
arithmetic/inequalities
arithmetic/rationals-with-axioms

The formal definition is then further built on the following.

A.1 Macros

INCLUDING the book: */slocal/src/acl2/v1-8/books/public/sets*.

Note the important difference between the entitys stack and the values stack. All entities, including variables will be contained in the entity stack. That is where we go to look up the value of a variable or constant, as well as where we get the body of a procedure or function.

MACRO:
mkenv (*entitys*, *values*, *assertions*)
=
`‘(list ,entitys ,values ,assertions)`

An entry can be a constant, variable, type, function, procedure, or package. The types of constants and variables are fully expanded types.

MACRO:
make-entry (&REST *entry*) = `‘(list ,@entry)`

MACRO:
entry-name (*x*) = `‘(car ,x)`

MACRO:
entry-decl (*x*) = `‘(nth 1 ,x)`

MACRO:
entry-value (*x*) = `‘(nth 2 ,x)`

Get the stack*.

MACRO:
es (*env*) = `‘(nth 0 ,env)`

entry stack

MACRO:
vs (*env*) = `‘(nth 1 ,env)`

value stack

MACRO:
as (*env*) = `‘(nth 2 ,env)`

assertion stack

Set the stack*.

MACRO:
set-es (*x*, *env*)
=
```
‘(list
  ,x
  (vs ,env)
  (as ,env))
```

MACRO:
set-vs (*x*, *env*)
=
```
‘(list
  (es ,env)
  ,x
  (as ,env))
```

MACRO:
set-as (*x*, *env*)
=
```
‘(list
  (es ,env)
  (vs ,env)
  ,x)
```

Return the top stack of the stack*.

MACRO:
top-es (*env*) = `‘(car (es ,env))`

MACRO:
top-vs (*env*) = `‘(car (vs ,env))`

MACRO:
top-as (*env*) = `‘(car (as ,env))`

Top value stack entry

MACRO:
top-vse (*env*) = `‘(car (top-vs ,env))`

MACRO:
top-value (*env*) = `‘(car (top-vs ,env))`

MACRO:
nth-vse (*n*, *env*)
=
```
‘(nth ,n (top-vs ,env))
```

Pop a stack from the stack*.

MACRO:
pop-es (*env*)
=
```
‘(list
  (cdr (es ,env))
  (vs ,env)
  (as ,env))
```

MACRO:

pop-vs (*env*)
 =
```
‘(list
  (es ,env)
  (cdr (vs ,env))
  (as ,env))
```

MACRO:
pop-as (*env*)
 =
```
‘(list
  (es ,env)
  (vs ,env)
  (cdr (as ,env)))
```

Push a stack onto the stack*.

MACRO:
push-es (*x*, *env*)
 =
```
‘(list
  (cons ,x (es ,env))
  (vs ,env)
  (as ,env))
```

MACRO:
push-vs (*x*, *env*)
 =
```
‘(list
  (es ,env)
  (cons ,x (vs ,env))
  (as ,env))
```

MACRO:
push-as (*x*, *env*)
 =
```
‘(list
  (es ,env)
  (vs ,env)
  (cons ,x (as ,env)))
```

Push an entry onto the the top stack of the stack*.

MACRO:
push-ese (*x*, *env*)
 =
```
‘(list
  (cons
   (cons ,x (top-es ,env))
   (cdr (es ,env)))
  (vs ,env)
  (as ,env))
```

MACRO:
push-vse (*x*, *env*)
 =
```
‘(list
  (es ,env)
  (cons
   (cons ,x (top-vs ,env))
```

```
   (cdr (vs ,env)))
  (as ,env))
```

MACRO:
push-ase (*x*, *env*)
=

```
‘(list
  (es ,env)
  (vs ,env)
  (cons
   (cons ,x (top-as ,env))
   (cdr (as ,env))))
```

MACRO:
pop-ese (*env*)
=

```
‘(list
  (cons
   (cdr (top-es ,env))
   (cdr (es ,env)))
  (vs ,env)
  (as ,env))
```

MACRO:
pop-vse (*env*)
=

```
‘(list
  (es ,env)
  (cons
   (cdr (top-vs ,env))
   (cdr (vs ,env)))
  (as ,env))
```

MACRO:
pop-ase (*env*)
=

```
‘(list
  (es ,env)
  (vs ,env)
  (cons
   (cdr (top-as ,env))
   (cdr (as ,env))))
```

MACRO:
append-vse (*x*, *env*)
=

```
‘(list
  (es ,env)
  (cons
   (append ,x (top-vs ,env))
   (cdr (vs ,env)))
  (as ,env))
```

MACRO:
append-ase (*x*, *env*)
=

```
‘(list
  (es ,env)
  (vs ,env)
  (cons
   (append ,x (top-as ,env))
```

```
   (cdr (as ,env))))
```

MACRO:
push-entry (*x*, *env*) = `‘(push-ese ,x ,env)`

MACRO:
push-value (*x*, *env*) = `‘(push-vse ,x ,env)`

MACRO:
push-assert (*x*, *env*) = `‘(push-ase ,x ,env)`

MACRO:
pop-value (*env*) = `‘(pop-vse ,env)`

MACRO:
push-env (&OPTIONAL *env* := `’env`)
=
```
‘(list
  (cons nil (es ,env))
  (cons nil (vs ,env))
  (cons nil (as ,env)))
```

MACRO:
pop-env (&OPTIONAL *env* := `’env`)
=
```
‘(list
  (cdr (es ,env))
  (cdr (vs ,env))
  (cdr (as ,env)))
```

DEFINITION:
update-v (*x*, *val*, *va*)
=
if atom (*va*) **then** *va*
elseif atom (car (*va*)) **then** cons (car (*va*), update-v (*x*, *val*, cdr (*va*)))
elseif *x* = caar (*va*) **then** cons (make-entry (*x*, cadr (*va*), *val*), cdr (*va*))
else cons (car (*va*), update-v (*x*, *val*, cdr (*va*)))
fi

DEFINITION:
set-vse* (*x*, *val*, *vs*)
=
if atom (*vs*) **then** *vs*
elseif assoc (*x*, car (*vs*)) **then** cons (update-v (*x*, *val*, car (*vs*)), cdr (*vs*))
else cons (car (*vs*), set-vse* (*x*, *val*, cdr (*vs*)))
fi

DEFINITION:
set-vse (*x*, *val*, *env*) = mkenv (env_e, set-vse* (*x*, *val*, env_v), env_a)

Other, non-env related, macros.

MACRO:
int-not-in-machine-range (*val*)
=
```
‘(int-not-in-range
  ,val
  (ava-min-int)
  (ava-max-int))
```

A.2 Ilet

SET CURRENT PACKAGE to be **ACL2**.

INCLUDING the book: *leval-macros*.

MACRO:
assert1 (*x*)
=
```
‘(mv nil (push-ase ,x env))
```

MACRO:
asserts (*l*)
=
```
‘(mv nil (append-ase ,l env))
```

MACRO:
check-assert (*form*,
&OPTIONAL
outstate := `’(es env)`,
instate := `’nil`
)
=
```
‘(check-annotations
  (list ,form)
  ,instate
  ,outstate
  env)
```

MACRO:
check-asserts (&OPTIONAL *outstate* := `’(es env)`, *instate* := `’nil`)
=
```
‘(check-annotations
  (top-as env)
  ,instate
  ,outstate
  env)
```

CONSTANT:
logical-error = `’logical-error`

MACRO:
hard-error2 (*fmt-string*, &REST *args*)
=
```
‘(list
  ’hard-error
  ,fmt-string
  (list ,@args))
```

MACRO:
hard-error-env (*env*, *fmt-string*, &REST *args*)
=
```
‘(mv
  (list
   ’hard-error
   ,fmt-string
   (list ,@args))
  ,env)
```

DEFINITION:
check-annotations (*l*, *instate*, *outstate*, *env*)

```
 =
if null (l)  then ⟨nil, env⟩
 elseif leval (car (l), outstate, instate)
 then check-annotations (cdr (l), instate, outstate, env)
 else ⟨*logical-error*, env⟩
fi
```

DEFINITION:

```
do-mv-macro (l)
 =
if ¬ consp (l)
 then if null (l)  then '(mv exc env)
        else ['mv, 'exc, l]
       fi
 elseif ¬ consp (car (l))
 then if cdr (l)
        then hard-error2 ("Atomic non-terminal element of ilet (~s)",
                          l)
        else `(mv nil ,(car l))
       fi
 elseif caar (l) = ':=
 then let* arg be car (l)_1,
              step be car (l)_2,
              fail be cdddr (car (l)) ∧ car (l)_3,
              rest be do-mv-macro (cdr (l))
              in
       if symbolp (arg)
        then if arg = 'exc
              then `(let
                         ((exc ,step))
                         (if exc
                          ,(if fail fail
                             '(mv exc env))
                          ,rest))
              elseif fail
              then hard-error2 ("~%Cannot have FAIL branch without EXC (~s)",
                                l)
              else `(let
                        ((,arg ,step))
                        ,rest)
             fi
        elseif consp (arg)
        then if 'exc ∈ arg
              then `(mv-let
                         ,arg
                         ,step
                         (if exc
                          ,(if fail fail
                             '(mv exc env))
                          ,rest))
              elseif fail
              then hard-error2 ("~%Cannot have FAIL branch without EXC (~s)",
                                l)
              else `(mv-let
                        ,arg
                        ,step
                        ,rest)
             fi
```

```
      else hard-error2 ("~%Arg to := must be symbol or list (~s)",
                        l)
      fi
elseif caar (l) = 'exception
then let* form be car (l)1,
          fail be if cddr (car (l))  then car (l)2
                  else nil
                  fi,
          rest be do-mv-macro (cdr (l))
          in
     '(let
       ((exc ,form))
       (if exc
        ,(if fail fail
           '(mv exc env))
        ,rest))
elseif null (cdr (l)) ∧ (caar (l) = 'mv)  then car (l)
else let form be car (l),
         rest be do-mv-macro (cdr (l))
         in
    '(mv-let
      (exc env)
      ,form
      (if exc
       (mv exc env)
       ,rest))
fi
```

MACRO:
ilet (&REST *l*)
=
do-mv-macro (*l*)

A.3 Trees for Array and Record Literals

In this version, labels can be integers, symbols or lists. In principle, there is no reason to limit them.

Leaves of trees can be integers, characters, strings, or the symbols TRUE or FALSE. We excluded other symbols because it was important to ensure that

treep(x) -> ¬ leafp(x) ∧
leafp(x) -> ¬ treep(x)

We probaly could just as well replace boolean with non-nil-symbol.

leaf == integer || character || boolean || string
tree == ((label . node)*)
node == leaf || tree
label == integer || symbol || consp

Note from previous versions :

1. We changed ADA-BOOLEANP so that NIL is not both leafp and treep.
2. We let LABELP include CONSP because we may want to leave record selectors as (id

field-name 23).

SET CURRENT PACKAGE to be **ACL2**.

DEFINITION:
ada-booleanp (x) = (x = `'true`) ∨ (x = `'false`)

DEFINITION:
leaf-p (x)
=
integerp (x) ∨ characterp (x) ∨ ada-booleanp (x) ∨ stringp (x)

DEFINITION:
labelp (x) = integerp (x) ∨ symbolp (x) ∨ consp (x)

DEFINITION:
treep (x)
=
if consp (x)
then consp (car (x))
∧ **let** *label* **be** car (car (x)),
node **be** cdr (car (x))
in
labelp (*label*) ∧ (leaf-p (*node*) ∨ treep (*node*))
∧ treep (cdr (x))
else null (x)
fi

DEFINITION:
labelp* (l)
=
if ¬ consp (l) **then** null (l)
elseif labelp (car (l)) **then** labelp* (cdr (l))
else nil
fi

Labelp theorems

THEOREM: labelp*-cdr *:forward-chaining, :rewrite*
labelp* (l) → labelp* (cdr (l))

THEOREM: labelp-car
labelp* (l) → labelp (car (l))

THEOREM: labelp*-singleton
labelp (i) → labelp* ([i])

THEOREM: labelp*-list-car-if-labelp*
(labelp* (l) ∧ l) → labelp* ([car (l)])

THEOREM: labelp*-true-listp *:compound-recognizer*
labelp* (l) → true-listp (l)

THEOREM: consp-labelp* *:forward-chaining*
(labelp* (l) ∧ l) → consp (l)

THEOREM: labelp*-fwd *:forward-chaining*
labelp* (cons (i, l)) → (labelp* (l) ∧ labelp (i))

THEOREM: labelp*-bkwd
(labelp* (l) ∧ labelp (i)) → labelp* (cons (i, l))

MODIFY the current theory:

Disable ‘labelp*’ and ‘labelp’.

`These THEOREMS must precede put-l, get-l`

THEOREM: treep-alistp *:forward-chaining*
treep $(x) \rightarrow$ alistp (x)

THEOREM: treep-cdr
(treep $(x) \wedge x) \rightarrow$ treep (cdr (x))

THEOREM: treep-nil
treep (**nil**)

THEOREM: treep-cdr-fwd *:forward-chaining*
treep (cons $(w, x)) \rightarrow$ treep (x)

MODIFY the current theory:

Disable ‘leaf-p’ and ‘treep’.

Valuep and theorems

DEFINITION:
valuep (x) = leaf-p $(x) \vee$ treep (x)

THEOREM: leaf-p-not-treep *:forward-chaining*
leaf-p $(x) \rightarrow (\neg$ treep $(x))$

THEOREM: treep-not-leaf-p *:forward-chaining*
treep $(x) \rightarrow (\neg$ leaf-p $(x))$

THEOREM: not-treep-imp-leaf-p
$(\neg$ treep $(x)) \rightarrow$ (valuep $(x) \leftrightarrow$ leaf-p (x))

THEOREM: not-leaf-p-imp-treep
$(\neg$ leaf-p $(x)) \rightarrow$ (valuep $(x) \leftrightarrow$ treep (x))

THEOREM: leaf-p-valuep *:forward-chaining*
leaf-p $(x) \rightarrow$ valuep (x)

THEOREM: treep-valuep *:forward-chaining*
treep $(x) \rightarrow$ valuep (x)

THEOREM: treep-cons
(treep $(z) \wedge$ labelp $(i) \wedge$ valuep (val))
$\rightarrow$ treep (cons (cons $(i, val), z$))

MODIFY the current theory:

Disable ‘valuep’.

Domain & range

The list of CARs of a tree.

DEFINITION:
domain (x) = strip-cars (x)
Guard: treep (x)

The list of CDRs of a tree.

DEFINITION:

range (*x*) = strip-cdrs (*x*)
Guard: treep (*x*)

MODIFY the current theory:

Enable ‘labelp’, ‘labelp*’ and ‘treep’.

A tree containing only those pairs in A whose CAR is in L.

DEFINITION:
domain-restrict (*l*, *a*)
=
if (¬ consp (*a*)) ∨ (¬ consp (car (*a*))) **then nil**
elseif member-equal (car (car (*a*)), *l*)
then cons (car (*a*), domain-restrict (*l*, cdr (*a*)))
else domain-restrict (*l*, cdr (*a*))
fi
Guard: labelp* (*l*) ∧ treep (*a*)

MODIFY the current theory:

Disable ‘labelp’, ‘labelp*’ and ‘treep’.

Atomic versions of GET and SET. Take a SINGLE label.

(get-t nil i) = nil (get-t x nil) = Treats NIL as symbol and looks for it.

MODIFY the current theory:

Enable ‘treep’.

DEFINITION:
get-t (*tree*, *i*)
=
if (¬ consp (*tree*)) ∨ (¬ consp (car (*tree*))) **then nil**
elseif *i* = car (car (*tree*)) **then** cdr (car (*tree*))
else get-t (cdr (*tree*), *i*)
fi
Guard: labelp (*i*) ∧ treep (*tree*)

(put-t nil i v) = ((i . v)) (put-t nil nil nil) = ((nil . nil))

DEFINITION:
put-t (*tree*, *i*, *val*)
=
if (¬ consp (*tree*)) ∨ (¬ consp (car (*tree*))) **then** [cons (*i*, *val*)]
elseif *i* = car (car (*tree*)) **then** cons (cons (*i*, *val*), cdr (*tree*))
else cons (car (*tree*), put-t (cdr (*tree*), *i*, *val*))
fi
Guard: treep (*tree*) ∧ labelp (*i*) ∧ valuep (*val*)

DEFINITION:
exists-t (*tree*, *i*)
=
if (¬ consp (*tree*)) ∨ (¬ consp (car (*tree*))) **then nil**
elseif *i* = car (car (*tree*)) **then t**
else exists-t (cdr (*tree*), *i*)
fi
Guard: treep (*tree*) ∧ labelp (*i*)

(defthm exists-in-domain (implies (and (treep tree) (labelp i)) (iff (exists-t tree i) (member-equal i (domain tree)))) :hints (("Goal" :in-theory (disable labelp))))

MODIFY the current theory:

Disable 'treep'.

(in-theory (disable exists-in-domain))

Get-t openers

THEOREM: get-t-opener-1
(null (*tree*) ∧ force (labelp (*i*))) → (get-t (*tree*, *i*) = **nil**)

THEOREM: get-t-opener-2
((*i* = car (car (*tree*)))
∧ *tree*
∧ force (labelp (*i*))
∧ force (treep (*tree*)))
→ (get-t (*tree*, *i*) = cdar (*tree*))

THEOREM: get-t-opener-3
((*i* ≠ car (car (*tree*)))
∧ *tree*
∧ force (labelp (*i*))
∧ force (treep (*tree*)))
→ (get-t (*tree*, *i*) = get-t (cdr (*tree*), *i*))

THEOREM: get-t-nil-2
((¬ exists-t (*x*, *i*)) ∧ force (labelp (*i*)) ∧ force (treep (*x*)))
→ (get-t (*x*, *i*) = **nil**)

PUT-T openers

THEOREM: put-t-opener-1
(null (*tree*)
∧ force (treep (*tree*))
∧ force (labelp (*i*))
∧ force (valuep (*val*)))
→ (put-t (*tree*, *i*, *val*) = [cons (*i*, *val*)])

THEOREM: put-t-opener-2
(*tree*
∧ (*i* = car (car (*tree*)))
∧ force (treep (*tree*))
∧ force (labelp (*i*))
∧ force (valuep (*val*)))
→ (put-t (*tree*, *i*, *val*) = cons (cons (*i*, *val*), cdr (*tree*)))

Acl2 count lemmas Count theorems about trees, needed to admit get-l and exists-l.

THEOREM: acl2-count-cadar-3
(car (*x*) ∧ treep (*x*) ∧ (0 ≤ *w*) ∧ integerp (*w*))
→ (acl2-count (cdar (*x*)) < (1 + acl2-count (car (*x*)) + *w*))

THEOREM: acl2-count-get-t-2
(exists-t (*x*, *i*) ∧ force (treep (*x*)) ∧ force (labelp (*i*)))
→ (acl2-count (get-t (*x*, *i*)) < acl2-count (*x*))

TREEP theorems

THEOREM: treep-cons-car
(treep (y) ∧ y) → treep ([car (y)])

THEOREM: treep-cons-cdr
(treep (x) ∧ treep (y) ∧ y) → treep (cons (car (y), x))

PUT-T theorems

THEOREM: treep-put-t
(force (treep (x)) ∧ force (labelp (i)) ∧ force (valuep (*val*)))
→ treep (put-t (x, i, *val*))

THEOREM: get-t-put-t-identity
(force (treep (x)) ∧ force (valuep (*foo*)) ∧ force (labelp (i)))
→ (get-t (put-t (x, i, *foo*), i) = *foo*)

List versions of GET and SET.

(exists-l nil i) = nil (get-l nil l) = nil (get-l x nil) = x

MODIFY the current theory:

Enable ‘treep’.

MODIFY the current theory:

Disable ‘leaf-p’.

THEOREM: exists-imp-not-leaf-p-get-treep-get
(labelp* (cons (*l1*, *l2*))
∧ treep (x)
∧ exists-t (x, *l1*)
∧ (¬ leaf-p (get-t (x, *l1*))))
→ treep (get-t (x, *l1*))

DEFINITION:
get-l (x, l)
=
if ¬ consp (l) **then** x
elseif leaf-p (x) **then nil**
elseif (¬ treep (x)) ∨ (¬ labelp (car (l))) **then nil**
elseif exists-t (x, car (l)) **then** get-l (get-t (x, car (l)), cdr (l))
else nil
fi
Guard: (leaf-p (x) ∨ treep (x)) ∧ labelp* (l)

MODIFY the current theory:

Disable ‘treep’.

(exists-l nil i) = nil (put-l x nil v) = v

DEFINITION:
put-l (x, l, *val*)
=
if ¬ consp (l) **then** *val*
elseif ¬ labelp (car (l)) **then** *val*
elseif ¬ treep (x) **then** put-t (**nil**, car (l), put-l (**nil**, cdr (l), *val*))
elseif null (cdr (l)) **then** put-t (x, car (l), *val*)
else put-t (x, car (l), put-l (get-l (x, [car (l)]), cdr (l), *val*))
fi

Guard: (leaf-p (*x*) ∨ treep (*x*)) ∧ labelp* (*l*) ∧ valuep (*val*)

(exists-l nil nil) = T (exists-l x nil) = T (exists-l nil (cons a b)) = F

DEFINITION:
exists-l (*tree*, *l*)
=
if ¬ consp (*l*) **then t**
elseif ¬ labelp (car (*l*)) **then nil**
elseif ¬ treep (*tree*) **then nil**
elseif exists-t (*tree*, car (*l*)) ∧ treep (get-t (*tree*, car (*l*)))
then exists-l (get-t (*tree*, car (*l*)), cdr (*l*))
else nil
fi
Guard: treep (*tree*) ∧ labelp* (*l*)

DEFINITION:
distinct (*i*, *j*)
=
if (¬ consp (*i*)) ∨ (¬ consp (*j*)) **then nil**
elseif car (*i*) ≠ car (*j*) **then t**
else distinct (cdr (*i*), cdr (*j*))
fi
Guard: labelp* (*i*) ∧ labelp* (*j*)

Put-l, Get-l Theorems

THEOREM: valuep-get-t
(force (treep (*x*)) ∧ force (labelp (*i*))) → valuep (get-t (*x*, *i*))

THEOREM: null-get-l-2
((¬ treep (*w*)) ∧ force (valuep (*w*)) ∧ force (labelp* (*l*)))
→ (get-l (*w*, *l*)
= **if** null (*l*) **then** *w*
else nil
fi)

THEOREM: valuep-get-l-1
((¬ treep (get-t (*x*, *i*))) ∧ labelp (*i*) ∧ treep (*x*))
→ leaf-p (get-t (*x*, *i*))

THEOREM: valuep-get-l
(force (treep (*x*)) ∧ force (labelp* (*l*))) → valuep (get-l (*x*, *l*))

THEOREM: valuep-put-t
(force (labelp (*i*)) ∧ force (treep (*w*)) ∧ force (valuep (*val*)))
→ valuep (put-t (*w*, *i*, *val*))

MODIFY the current theory:

Enable ‘valuep’.

THEOREM: valuep-get-l-list-i
(treep (*m*) ∧ labelp (*i*)) → valuep (get-l (*m*, [*i*]))

THEOREM: valuep-put-l
(force (labelp* (*l*)) ∧ force (valuep (*w*)) ∧ force (valuep (*val*)))
→ valuep (put-l (*w*, *l*, *val*))

THEOREM: treep-put-l
(*l2* ∧ force (labelp* (*l2*)) ∧ force (valuep (*w*)) ∧ force (valuep (*val*)))

$\rightarrow$ treep (put-l (w, $l2$, val))

THEOREM: valuep-cons-put-l-nil
(valuep (val) $\land$ labelp* (l) $\land$ l)
$\rightarrow$ valuep ([cons (car (l), put-l (**nil**, cdr (l), val))])

MODIFY the current theory:

Disable ‘valuep’, ‘leaf-p’ and ‘treep’.

VERIFY GUARDS for ‘put-l’

THEOREM: leaf-p-not-nil *:forward-chaining*
leaf-p (val) $\rightarrow$ val

THEOREM: not-leaf-p-cons
$\neg$ leaf-p (cons (a, b))

THEOREM: put-l-leaf-p
(l
$\land$ ($\neg$ treep (w))
$\land$ force (labelp* (l))
$\land$ force (valuep (val))
$\land$ force (valuep (w)))
$\rightarrow$ (put-l (w, l, val) = put-l (**nil**, l, val))

THEOREM: exists-if-put-t
(force (labelp (i)) $\land$ force (valuep (w)) $\land$ force (treep (x)))
$\rightarrow$ exists-t (put-t (x, i, w), i)

THEOREM: not-leaf-p-put-t
(labelp (i) $\land$ valuep (val) $\land$ treep (x))
$\rightarrow$ ($\neg$ leaf-p (put-t (x, i, val)))

THEOREM: get-put-l
(l $\land$ force (treep (x)) $\land$ force (valuep (val)) $\land$ force (labelp* (l)))
$\rightarrow$ (get-l (put-l (x, l, val), l) = val)

THEOREM: treep-cons-put-t
(x $\land$ force (treep (x)) $\land$ force (labelp (i)) $\land$ force (valuep (w)))
$\rightarrow$ treep (cons (car (x), put-t (cdr (x), i, w)))

THEOREM: get-t-put-t-ne
(($i \neq j$)
$\land$ get-t (x, j)
$\land$ force (labelp (j))
$\land$ force (labelp (i))
$\land$ force (treep (x))
$\land$ force (valuep (w)))
$\rightarrow$ (get-t (put-t (x, i, w), j) = get-t (x, j))

THEOREM: not-get-t-step
(($\neg$ get-t (x, j))
$\land$ ($i \neq j$)
$\land$ force (treep (x))
$\land$ force (valuep (w))
$\land$ force (labelp (i))
$\land$ force (labelp (j)))
$\rightarrow$ ($\neg$ get-t (put-t (x, i, w), j))

THEOREM: fail-get-l

$((\neg \text{get-t}(x, \text{car}(l))) \land l \land \text{force}(\text{treep}(x)) \land \text{force}(\text{labelp*}(l)))$
$\rightarrow (\text{get-l}(x, l) = \textbf{nil})$

THEOREM: put-t-opener-3

$$\begin{aligned}
&(&& (i \neq \text{car}(\text{car}(tree))) \\
&\land && \text{consp}(tree) \\
&\land && \text{consp}(\text{car}(tree)) \\
&\land && \text{force}(\text{treep}(tree)) \\
&\land && \text{force}(\text{labelp}(i)) \\
&\land && \text{force}(\text{valuep}(val))) \\
&\rightarrow && (\text{put-t}(tree, i, val) = \text{cons}(\text{car}(tree), \text{put-t}(\text{cdr}(tree), i, val)))
\end{aligned}$$

THEOREM: get-l-opener-1
$(\text{null}(l) \land (\text{leaf-p}(x) \lor \text{treep}(x)) \land \text{force}(\text{labelp*}(l)))$
$\rightarrow (\text{get-l}(x, l) = x)$

THEOREM: get-l-opener-3
$(l \land \text{exists-t}(x, \text{car}(l)) \land \text{treep}(x) \land \text{force}(\text{labelp*}(l)))$
$\rightarrow (\text{get-l}(x, l) = \text{get-l}(\text{get-t}(x, \text{car}(l)), \text{cdr}(l)))$

THEOREM: get-l-opener-4
$(l \land (\neg \text{exists-t}(x, \text{car}(l))) \land \text{treep}(x) \land \text{force}(\text{labelp*}(l)))$
$\rightarrow (\text{get-l}(x, l) = \textbf{nil})$

THEOREM: put-l-opener-1

$$\begin{aligned}
&(&& \text{null}(l) \\
&\land && (\text{leaf-p}(x) \lor \text{treep}(x)) \\
&\land && \text{force}(\text{labelp*}(l)) \\
&\land && \text{force}(\text{valuep}(val))) \\
&\rightarrow && (\text{put-l}(x, l, val) = val)
\end{aligned}$$

THEOREM: put-l-opener-2
$(l \land \text{leaf-p}(x) \land \text{force}(\text{labelp*}(l)) \land \text{force}(\text{valuep}(val)))$
$\rightarrow (\text{put-l}(x, l, val) = \text{put-t}(\textbf{nil}, \text{car}(l), \text{put-l}(\textbf{nil}, \text{cdr}(l), val)))$

THEOREM: put-l-opener-3

$$\begin{aligned}
&(&& l \\
&\land && \text{null}(\text{cdr}(l)) \\
&\land && \text{treep}(x) \\
&\land && \text{force}(\text{labelp*}(l)) \\
&\land && \text{force}(\text{valuep}(val))) \\
&\rightarrow && (\text{put-l}(x, l, val) = \text{put-t}(x, \text{car}(l), val))
\end{aligned}$$

THEOREM: put-l-opener-4

$$\begin{aligned}
&(&& l \\
&\land && (\neg \text{null}(\text{cdr}(l))) \\
&\land && \text{treep}(x) \\
&\land && \text{force}(\text{labelp*}(l)) \\
&\land && \text{force}(\text{valuep}(val))) \\
&\rightarrow (&& \text{put-l}(x, l, val) \\
&= && \text{put-t}(x, \text{car}(l), \text{put-l}(\text{get-l}(x, [\text{car}(l)]), \text{cdr}(l), val)))
\end{aligned}$$

THEOREM: exists-t-opener-1
$(\text{null}(tree) \land \text{force}(\text{treep}(tree)) \land \text{force}(\text{labelp}(i)))$
$\rightarrow (\text{exists-t}(tree, i) = \textbf{nil})$

THEOREM: exists-t-opener-2

$$\begin{aligned}
&(&& \text{consp}(tree) \\
&\land && \text{consp}(\text{car}(tree)) \\
&\land && (i = \text{car}(\text{car}(tree))) \\
&\land && \text{force}(\text{treep}(tree))
\end{aligned}$$

$\wedge$ force (labelp (i)))
$\rightarrow$ (exists-t ($tree$, i) = **t**)

THEOREM: exists-t-opener-3
(consp ($tree$)
$\wedge$ consp (car ($tree$))
$\wedge$ ($i \neq$ car (car ($tree$)))
$\wedge$ force (treep ($tree$))
$\wedge$ force (labelp (i)))
$\rightarrow$ (exists-t ($tree$, i) = exists-t (cdr ($tree$), i))

MODIFY the current theory:

Disable 'get-t', 'get-l', 'put-t' and 'put-l'.

THEOREM: exists-t-exists-l
($l \wedge$ exists-l (x, l) $\wedge$ force (treep (x)) $\wedge$ force (labelp* (l)))
$\rightarrow$ exists-t (x, car (l))

THEOREM: exists-t-put-t
(exists-t (x, j)
$\wedge$ ($i \neq j$)
$\wedge$ force (treep (x))
$\wedge$ force (valuep (w))
$\wedge$ force (labelp (i))
$\wedge$ force (labelp (j)))
$\rightarrow$ exists-t (put-t (x, i, w), j)

THEOREM: get-l-put-t-opener
(l
$\wedge$ ($i \neq$ car (l))
$\wedge$ force (treep (x))
$\wedge$ force (labelp* (l))
$\wedge$ force (valuep (w))
$\wedge$ force (labelp (i)))
$\rightarrow$ (get-l (put-t (x, i, w), l)
= **if** get-t (x, car (l)) **then** get-l (x, l)
else nil
fi)

DEFINITION:
induct3 (i, j, x)
=
if ($\neg$ consp (i)) $\vee$ ($\neg$ consp (j)) **then nil**
elseif car (i) $\neq$ car (j) **then nil**
elseif leaf-p (x) **then nil**
elseif null (cdr (j)) **then nil**
elseif exists-t (x, car (i)) **then** induct3 (cdr (i), cdr (j), get-t (x, car (i)))
else induct3 (cdr (i), cdr (j), get-t (x, car (i)))
fi
Guard: (leaf-p (x) $\vee$ treep (x)) $\wedge$ labelp* (i) $\wedge$ labelp* (j)

THEOREM: not-distinct
(labelp* (i)
$\wedge$ labelp* (j)
$\wedge$ i
$\wedge$ j
$\wedge$ (car (i) = car (j))
$\wedge$ ($\neg$ cdr (j)))
$\rightarrow$ ($\neg$ distinct (i, j))

THEOREM: get-l-put-l-opener-unequal-cars
(labelp* (i)
∧ labelp* (j)
∧ i
∧ j
∧ (car (i) ≠ car (j))
∧ treep (x)
∧ valuep (val))
→ (get-l (put-l (x, i, val), j) = get-l (x, j))

THEOREM: get-l-put-l-nil
(distinct (i, j)
∧ force (labelp* (i))
∧ force (labelp* (j))
∧ force (valuep (val)))
→ (get-l (put-l (**nil**, i, val), j) = get-l (**nil**, j))

THEOREM: not-get-l-leaf
(l ∧ (¬ treep (w)) ∧ labelp* (l) ∧ force (valuep (w)))
→ (¬ get-l (w, l))

THEOREM: get-put-i-j
(distinct (i, j)
∧ force (treep (x))
∧ force (valuep (val))
∧ force (labelp* (i))
∧ force (labelp* (j)))
→ (get-l (put-l (x, i, val), j) = get-l (x, j))

A.4 Support for Tree Equality Reasoning

SET CURRENT PACKAGE to be **ACL2**.

INCLUDING the book: *get-tree*.

MODIFY the current theory: Disable ‘leaf-p’, ‘labelp’, their executable counterparts.

DEFINITION:
lessp-or-equal (x, y)
=
if $x = y$ **then t**
elseif rationalp (x)
then if rationalp (y) **then** $x < y$
else t
fi
elseif rationalp (y) **then nil**
elseif characterp (x)
then if characterp (y) **then** char< (x, y)
else t
fi
elseif characterp (y) **then nil**
elseif symbolp (x)
then if symbolp (y) **then** symbol-< (x, y)
else t
fi
elseif symbolp (y) **then nil**
elseif stringp (x)
then if stringp (y) **then** string< (x, y)

else t
fi
elseif stringp (y) **then nil**
elseif $\neg$ consp (x) **then t**
elseif $\neg$ consp (y) **then nil**
elseif car (x) = car (y) **then** lessp-or-equal (cdr (x), cdr (y))
elseif lessp-or-equal (car (x), car (y)) **then t**
else lessp-or-equal (cdr (x), cdr (y))
fi

DEFINITION:
insert (a, l)
=
if $\neg$ consp (l) **then** cons (a, l)
elseif consp (car (l))
$\wedge$ consp (a)
$\wedge$ lessp-or-equal (car (car (l)), car (a))
then cons (car (l), insert (a, cdr (l)))
else cons (a, l)
fi

DEFINITION:
sortl (*alist*)
=
if $\neg$ consp (*alist*) **then** *alist*
elseif $\neg$ consp (cdr (*alist*)) **then** *alist*
else insert (car (*alist*), sortl (cdr (*alist*)))
fi

Needed for ava-equal-1 admission below.

THEOREM: insert-cons-count=
acl2-count (insert (a, w)) = acl2-count (cons (a, w))

THEOREM: sortl-count=sup-1
acl2-count (insert (x, cx)) = (1 + acl2-count (x) + acl2-count (cx))

THEOREM: sortl-count=sup-2
(acl2-count (w) = m)
$\rightarrow$ (acl2-count (insert (a, w)) = (1 + acl2-count (a) + m))

THEOREM: true-listp-sortl
true-listp (x) $\rightarrow$ true-listp (sortl (x))

THEOREM: sortl-count=sup-3
(consp (x)
$\wedge$ consp (cdr (x))
$\wedge$ (acl2-count (sortl (cdr (x))) = acl2-count (cdr (x)))
$\wedge$ true-listp (cdr (x)))
$\rightarrow$ (acl2-count (insert (car (x), sortl (cdr (x))))
= (1 + acl2-count (car (x)) + acl2-count (cdr (x))))

THEOREM: sortl-count=
acl2-count (sortl (x)) = acl2-count (x)

THEOREM: strip-cdrs-count-le *:linear*
acl2-count (strip-cdrs (w)) $\leq$ acl2-count (w)

THEOREM: strip-cdrs-count-le-sup-1 *:linear*
(consp (w) $\wedge$ (acl2-count (w) $\leq$ acl2-count (x)))
$\rightarrow$ (acl2-count (cdr (w)) $<$ acl2-count (x))

THEOREM: strip-cdrs-count-le-sup-2 *:linear*
true-listp (x)
→ (acl2-count (strip-cdrs (sortl (cdr (x)))) ≤ acl2-count (x))

THEOREM: cdr-strip-cdrs-count-le
(true-listp (x) ∧ consp (x))
→ (acl2-count (cdr (strip-cdrs (sortl (cdr (x))))) < acl2-count (x))

THEOREM: treep-imp-true-listp *:forward-chaining*
treep (x) → true-listp (x)

MODIFY the current theory:

Enable ‘treep’.

THEOREM: simple-strip-count=1 *:linear*
(consp (w) ∧ treep (w))
→ (acl2-count (strip-cdrs (w)) < acl2-count (w))

THEOREM: treep-sortl=1
consp (x) → consp (sortl (x))

DEFINITION:
branchp (a)
=
labelp (car (a)) ∧ (leaf-p (cdr (a)) ∨ treep (cdr (a)))

THEOREM: treep-sortl=2-1
(($a \in x$) ∧ treep (x)) → branchp (a)

THEOREM: member-insert=1
member-equal (a, w) → member-equal (a, insert (z, w))

THEOREM: member-sortl=2
member-equal (a, x) → member-equal (a, sortl (x))

MODIFY the current theory:

Enable ‘treep’ and ‘leaf-p’.

THEOREM: sort-strip-count=1-a
(consp (sortl (x)) ∧ treep (sortl (x)))
→ (acl2-count (strip-cdrs (sortl (x))) < acl2-count (sortl (x)))

THEOREM: treep-insert
(consp (a) ∧ labelp (car (a)) ∧ treep (cdr (a)) ∧ treep (w))
→ treep (insert (a, w))

THEOREM: treep-sortl
treep (x) → treep (sortl (x))

THEOREM: sort-strip-count=1
(consp (x) ∧ treep (x))
→ (acl2-count (strip-cdrs (sortl (x))) < acl2-count (sortl (x)))

THEOREM: strip-cdrs-sort-consp-count
(consp (x) ∧ treep (x))
→ (acl2-count (strip-cdrs (sortl (x))) < acl2-count (x))

THEOREM: treep-sortl=2
treep (x) → treep (sortl (x))

A.5 Normalized Abstract Prefix

Input to the interpreter satisfies the predicate, top-prefix-p. The following functions define recognizers, constructors, and extractors for the various elements of the abstract prefix.

SET CURRENT PACKAGE to be **ACL2**.

INCLUDING the book: *kernel-extension*.

INCLUDING the book: *get-tree*.

INCLUDING the book: *subprefix-macros*.

DEFINITION:
symbolsp (*l*)
=
if consp (*l*)
then if symbolp (car (*l*)) **then** symbolsp (cdr (*l*))
else nil
fi
else *l* = **nil**
fi

DEFINITION:
arg1 (*p*)
=
if consp (*p*) **then** p_1
else nil
fi

DEFINITION:
arg2 (*p*)
=
if consp (*p*) **then** p_2
else nil
fi

DEFINITION:
arg3 (*p*)
=
if consp (*p*) **then** p_3
else nil
fi

DEFINITION:
arg4 (*p*)
=
if consp (*p*) **then** p_4
else nil
fi

DEFINITION:
arg5 (*p*)
=
if consp (*p*) **then** p_5
else nil
fi

DEFINITION:
arg6 (*p*)
=

if consp (p) **then** p_6
else nil
fi

DEFINITION:
arg7 (p)
=
if consp (p) **then** p_7
else nil
fi

DEFINITION:
arg8 (p)
=
if consp (p) **then** p_8
else nil
fi

DEFINITION:
arg9 (p)
=
if consp (p) **then** p_9
else nil
fi

DEFINITION:
ada-char-p (*form*)
=
characterp (*form*) ∧ standard-char-p (*form*)

DEFINITION:
acl2-boolean (*form*) = (*form* $=_{eq}$ `'t`) ∨ (*form* $=_{eq}$ **nil**)

DEFINITION:
all-integer (l)
=
if null (l) **then t**
elseif ¬ consp (l) **then nil**
elseif integerp (car (l)) **then** all-integer (cdr (l))
else nil
fi

DEFINITION:
all-id (l)
=
if null (l) **then t**
elseif ¬ consp (l) **then nil**
elseif listp (car (l)) ∧ (caar (l) $=_{eq}$ `'id`) **then** all-id (cdr (l))
else nil
fi

DEFINITION:
all-labels-integer (l) = labelp* (l) ∧ all-integer (l)

DEFINITION:
all-labels-id (l) = labelp* (l) ∧ all-id (l)

DEFINITION:
pre-type (x, *kind*)
=
(listp (x) ∧ (car (x) = `'id`))
∧ (cadr (x) = *kind*)

$\wedge$ (caddr (x) = `0`)

DEFINITION:
pre-integer (x) = pre-type (x, `'integer`)

DEFINITION:
pre-positive (x) = pre-type (x, `'positive`)

DEFINITION:
pre-natural (x) = pre-type (x, `'natural`)

DEFINITION:
pre-character (x) = pre-type (x, `'character`)

DEFINITION:
pre-string (x) = pre-type (x, `'string`)

DEFINITION:
pre-boolean (x) = pre-type (x, `'boolean`)

DEFINITION:
array-literal-p (*form*)
=
form $\wedge$ treep (*form*) $\wedge$ all-labels-integer (domain (*form*))

DEFINITION:
record-literal-p (*form*)
=
form $\wedge$ treep (*form*) $\wedge$ all-labels-id (domain (*form*))

DEFINITION:
error-p (x) = listp (x) $\wedge$ (car (x) = `'error`)

DEFINITION:
mk-error (*form*, *message*) = cons (`'error`, [*form*, *message*])

DEFINITION:
error-form (*form*) = arg* $(form)_{0}$

DEFINITION:
error-message (*form*) = arg* $(form)_{1}$

DEFINITION:
others-p (x) = consp (x) $\wedge$ (car (x) = `'others`)

DEFINITION:
mk-others = [`'others`]

DEFINITION:
unconstrained-p (x)
=
consp (x) $\wedge$ (car (x) = `'unconstrained`)

DEFINITION:
mk-unconstrained = [`'unconstrained`]

DEFINITION:
id-p (x) = listp (x) $\wedge$ (car (x) = `'id`)

DEFINITION:
mk-id (*root*, *uid*) = cons (`'id`, [*root*, *uid*])

DEFINITION:
id-root (*form*) = arg* $(form)_{0}$

DEFINITION:
$\text{id-uid}\,(\textit{form}) = \text{arg*}\,(\textit{form})_1$

DEFINITION:
$\text{true-p}\,(x) = \text{consp}\,(x) \wedge (\text{car}\,(x) = \texttt{'true})$

DEFINITION:
$\text{mk-true} = [\texttt{'true}]$

DEFINITION:
$\text{false-p}\,(x) = \text{consp}\,(x) \wedge (\text{car}\,(x) = \texttt{'false})$

DEFINITION:
$\text{mk-false} = [\texttt{'false}]$

DEFINITION:
$\text{boolean-literal-p}\,(x) = \text{true-p}\,(x) \vee \text{false-p}\,(x)$

DEFINITION:
$\text{list-p}\,(x) = \text{consp}\,(x) \wedge (\text{car}\,(x) = \texttt{'list})$

DEFINITION:
$\text{mk-list}\,(\textit{args}) = \text{cons}\,(\texttt{'list}, \textit{args})$

DEFINITION:
$$\begin{aligned}
&\text{literal-p}\,(x)\\
&=\\
&\quad \text{boolean-literal-p}\,(x)\\
&\vee\ (\quad \text{integerp}\,(x)\\
&\qquad \vee\ (\quad \text{stringp}\,(x)\\
&\qquad\qquad \vee\ (\quad \text{ada-char-p}\,(x)\\
&\qquad\qquad\qquad \vee\ (\text{array-literal-p}\,(x) \vee \text{record-literal-p}\,(x)))))
\end{aligned}$$

DEFINITION:
$$\begin{aligned}
&\text{lliteral-p}\,(x)\\
&=\\
&\text{literal-p}\,(x) \vee (\text{acl2-boolean}\,(x) \vee \text{list-p}\,(x))
\end{aligned}$$

DEFINITION:
$\text{enumeration-literal-p}\,(x) = \text{id-p}\,(x) \vee \text{ada-char-p}\,(x)$

DEFINITION:
$$\begin{aligned}
&\text{enumeration-p}\,(x)\\
&=\\
&\text{consp}\,(x) \wedge (\text{car}\,(x) = \texttt{'enumeration})
\end{aligned}$$

DEFINITION:
$\text{mk-enumeration}\,(\textit{args}) = \text{cons}\,(\texttt{'enumeration}, \textit{args})$

DEFINITION:
$$\begin{aligned}
&\text{predefined-type-p}\,(x)\\
&=\\
&\quad \text{pre-character}\,(x)\\
&\vee\ (\quad \text{pre-integer}\,(x)\\
&\qquad \vee\ (\quad \text{pre-positive}\,(x)\\
&\qquad\qquad \vee\ (\text{pre-natural}\,(x) \vee (\text{pre-string}\,(x) \vee \text{pre-boolean}\,(x)))))
\end{aligned}$$

DEFINITION:
$$\begin{aligned}
&\text{type-mark-p}\,(x)\\
&=\\
&\text{listp}\,(x) \wedge (\text{car}\,(x) = \texttt{'type-mark})
\end{aligned}$$

DEFINITION:
$\text{mk-type-mark}\,(\textit{type}, \textit{constraint})$

=
cons ('`type-mark`, [*type*, *constraint*])

DEFINITION:
type-mark-type (*form*) = arg* (*form*)$_0$

DEFINITION:
type-mark-constraint (*form*) = arg* (*form*)$_1$

DEFINITION:
subtype-p (x) = id-p (x) ∨ type-mark-p (x)

DEFINITION:
attribute-p (x)
=
listp (x) ∧ (car (x) = '`attribute`)

DEFINITION:
mk-attribute (*root*, *attr*) = cons ('`attribute`, [*root*, *attr*])

DEFINITION:
attribute-root (*form*) = arg* (*form*)$_0$

DEFINITION:
attribute-attr (*form*) = arg* (*form*)$_1$

DEFINITION:
range-p (x) = listp (x) ∧ (car (x) = '`range`)

DEFINITION:
mk-range (*from*, *to*) = cons ('`range`, [*from*, *to*])

DEFINITION:
range-from (*form*) = arg* (*form*)$_0$

DEFINITION:
range-to (*form*) = arg* (*form*)$_1$

DEFINITION:
constraint-p (x)
=
subtype-p (x)
∨ (unconstrained-p (x) ∨ (range-p (x) ∨ attribute-p (x)))

DEFINITION:
field-spec-p (x)
=
listp (x) ∧ (car (x) = '`field-spec`)

DEFINITION:
mk-field-spec (*id*, *decl*) = cons ('`field-spec`, [*id*, *decl*])

DEFINITION:
field-spec-id (*form*) = arg* (*form*)$_0$

DEFINITION:
field-spec-decl (*form*) = arg* (*form*)$_1$

DEFINITION:
record-type-p (x)
=
consp (x) ∧ (car (x) = '`record-type`)

DEFINITION:
mk-record-type (*args*) = cons ('`record-type`, *args*)

DEFINITION:
array-type-p (x)
=
listp (x) $\wedge$ (car (x) = `'array-type`)

DEFINITION:
mk-array-type (*index*, *elements*)
=
cons (`'array-type`, [*index*, *elements*])

DEFINITION:
array-type-index (*form*) = arg* $(form)_0$

DEFINITION:
array-type-elements (*form*) = arg* $(form)_1$

DEFINITION:
type-p (x)
=
record-type-p (x)
$\vee$ (array-type-p (x) $\vee$ (id-p (x) $\vee$ (range-p (x) $\vee$ enumeration-p (x))))

DEFINITION:
type-decl-p (x)
=
listp (x) $\wedge$ (car (x) = `'type-decl`)

DEFINITION:
mk-type-decl (*id*, *decl*) = cons (`'type-decl`, [*id*, *decl*])

DEFINITION:
type-decl-id (*form*) = arg* $(form)_0$

DEFINITION:
type-decl-decl (*form*) = arg* $(form)_1$

DEFINITION:
subtype-decl-p (x)
=
listp (x) $\wedge$ (car (x) = `'subtype-decl`)

DEFINITION:
mk-subtype-decl (*id*, *decl*) = cons (`'subtype-decl`, [*id*, *decl*])

DEFINITION:
subtype-decl-id (*form*) = arg* $(form)_0$

DEFINITION:
subtype-decl-decl (*form*) = arg* $(form)_1$

DEFINITION:
qualified-p (x)
=
listp (x) $\wedge$ (car (x) = `'qualified`)

DEFINITION:
mk-qualified (*type*, *value*) = cons (`'qualified`, [*type*, *value*])

DEFINITION:
qualified-type (*form*) = arg* $(form)_0$

DEFINITION:
qualified-value (*form*) = arg* $(form)_1$

DEFINITION:
type-convert-p (x)
=
listp (x) $\wedge$ (car (x) = `'type-convert`)

DEFINITION:
mk-type-convert (*type*, *value*)
=
cons (`'type-convert`, [*type*, *value*])

DEFINITION:
type-convert-type (*form*) = arg* (*form*)$_0$

DEFINITION:
type-convert-value (*form*) = arg* (*form*)$_1$

DEFINITION:
aggregate-choice-p (x)
=
listp (x) $\wedge$ (car (x) = `'aggregate-choice`)

DEFINITION:
mk-aggregate-choice (*choice*, *value*)
=
cons (`'aggregate-choice`, [*choice*, *value*])

DEFINITION:
aggregate-choice-choice (*form*) = arg* (*form*)$_0$

DEFINITION:
aggregate-choice-value (*form*) = arg* (*form*)$_1$

DEFINITION:
aggregate-pos-p (x)
=
listp (x) $\wedge$ (car (x) = `'aggregate-pos`)

DEFINITION:
mk-aggregate-pos (*value*) = cons (`'aggregate-pos`, [*value*])

DEFINITION:
aggregate-pos-value (*form*) = arg* (*form*)$_0$

DEFINITION:
aggregate-arm-p (x)
=
aggregate-choice-p (x) $\vee$ aggregate-pos-p (x)

DEFINITION:
aggregate-p (x)
=
consp (x) $\wedge$ (car (x) = `'aggregate`)

DEFINITION:
mk-aggregate (*args*) = cons (`'aggregate`, *args*)

DEFINITION:
indexed-component-p (x)
=
listp (x) $\wedge$ (car (x) = `'indexed-component`)

DEFINITION:
mk-indexed-component (*root*, *index*)
=

cons ('`indexed-component`, [*root*, *index*])

DEFINITION:
indexed-component-root (*form*) = arg* (*form*)$_0$

DEFINITION:
indexed-component-index (*form*) = arg* (*form*)$_1$

DEFINITION:
apply-1-p (*x*) = listp (*x*) $\wedge$ (car (*x*) = '`apply-1`)

DEFINITION:
mk-apply-1 (*root*, *args*) = cons ('`apply-1`, [*root*, *args*])

DEFINITION:
apply-1-root (*form*) = arg* (*form*)$_0$

DEFINITION:
apply-1-args (*form*) = arg* (*form*)$_1$

DEFINITION:
selected-component-p (*x*)
=
listp (*x*) $\wedge$ (car (*x*) = '`selected-component`)

DEFINITION:
mk-selected-component (*root*, *field*)
=
cons ('`selected-component`, [*root*, *field*])

DEFINITION:
selected-component-root (*form*) = arg* (*form*)$_0$

DEFINITION:
selected-component-field (*form*) = arg* (*form*)$_1$

DEFINITION:
designator-p (*x*)
=
consp (*x*) $\wedge$ (car (*x*) = '`designator`)

DEFINITION:
mk-designator (*args*) = cons ('`designator`, *args*)

DEFINITION:
dot-qual-1-p (*x*)
=
listp (*x*) $\wedge$ (car (*x*) = '`dot-qual-1`)

DEFINITION:
mk-dot-qual-1 (*root*, *component*)
=
cons ('`dot-qual-1`, [*root*, *component*])

DEFINITION:
dot-qual-1-root (*form*) = arg* (*form*)$_0$

DEFINITION:
dot-qual-1-component (*form*) = arg* (*form*)$_1$

DEFINITION:
defining-name-p (*x*) = id-p (*x*) $\vee$ designator-p (*x*)

DEFINITION:
name-p (*x*)

=
    id-p (x)
∨ (indexed-component-p (x)
    ∨ (selected-component-p (x) ∨ (apply-1-p (x) ∨ dot-qual-1-p (x))))

DEFINITION:
apl-p (x) = consp (x) ∧ (car (x) = `'apl`)

DEFINITION:
mk-apl ($args$) = cons (`'apl`, $args$)

DEFINITION:
op-expr-p (x) = listp (x) ∧ (car (x) = `'op-expr`)

DEFINITION:
mk-op-expr (id, $actuals$) = cons (`'op-expr`, [id, $actuals$])

DEFINITION:
op-expr-id ($form$) = arg* $(form)_0$

DEFINITION:
op-expr-actuals ($form$) = arg* $(form)_1$

DEFINITION:
function-call-p (x)
=
listp (x) ∧ (car (x) = `'function-call`)

DEFINITION:
mk-function-call (id, $actuals$)
=
cons (`'function-call`, [id, $actuals$])

DEFINITION:
function-call-id ($form$) = arg* $(form)_0$

DEFINITION:
function-call-actuals ($form$) = arg* $(form)_1$

DEFINITION:
expr-p (x)
=
    literal-p (x)
∨ (aggregate-p (x)
    ∨ (name-p (x)
        ∨ (op-expr-p (x)
            ∨ (function-call-p (x)
                ∨ (type-convert-p (x) ∨ qualified-p (x))))))

DEFINITION:
instate-p (x) = listp (x) ∧ (car (x) = `'instate`)

DEFINITION:
mk-instate ($expr$) = cons (`'instate`, [$expr$])

DEFINITION:
instate-expr ($form$) = arg* $(form)_0$

DEFINITION:
outstate-p (x) = listp (x) ∧ (car (x) = `'outstate`)

DEFINITION:
mk-outstate ($expr$) = cons (`'outstate`, [$expr$])

DEFINITION:
outstate-expr (*form*) = arg* (*form*)$_0$

DEFINITION:
assert-p (*x*) = listp (*x*) ∧ (car (*x*) = `'assert`)

DEFINITION:
mk-assert (*relation*) = cons (`'assert`, [*relation*])

DEFINITION:
assert-relation (*form*) = arg* (*form*)$_0$

DEFINITION:
invariant-p (*x*)
=
listp (*x*) ∧ (car (*x*) = `'invariant`)

DEFINITION:
mk-invariant (*relation*) = cons (`'invariant`, [*relation*])

DEFINITION:
invariant-relation (*form*) = arg* (*form*)$_0$

DEFINITION:
transition-p (*x*)
=
listp (*x*) ∧ (car (*x*) = `'transition`)

DEFINITION:
mk-transition (*relation*) = cons (`'transition`, [*relation*])

DEFINITION:
transition-relation (*form*) = arg* (*form*)$_0$

DEFINITION:
return-relation-p (*x*)
=
listp (*x*) ∧ (car (*x*) = `'return-relation`)

DEFINITION:
mk-return-relation (*var*, *relation*)
=
cons (`'return-relation`, [*var*, *relation*])

DEFINITION:
return-relation-var (*form*) = arg* (*form*)$_0$

DEFINITION:
return-relation-relation (*form*) = arg* (*form*)$_1$

DEFINITION:
return-value-p (*x*)
=
listp (*x*) ∧ (car (*x*) = `'return-value`)

DEFINITION:
mk-return-value (*relation*)
=
cons (`'return-value`, [*relation*])

DEFINITION:
return-value-relation (*form*) = arg* (*form*)$_0$

DEFINITION:

defaxiom-p (x) = listp $(x) \wedge$ (car (x) = $'\texttt{defaxiom}$)

DEFINITION:
mk-defaxiom $(id, relation)$ = cons $('\texttt{defaxiom}, [id, relation])$

DEFINITION:
defaxiom-id $(form)$ = arg* $(form)_0$

DEFINITION:
defaxiom-relation $(form)$ = arg* $(form)_1$

DEFINITION:
defthm-p (x) = listp $(x) \wedge$ (car (x) = $'\texttt{defthm}$)

DEFINITION:
mk-defthm $(id, relation)$ = cons $('\texttt{defthm}, [id, relation])$

DEFINITION:
defthm-id $(form)$ = arg* $(form)_0$

DEFINITION:
defthm-relation $(form)$ = arg* $(form)_1$

DEFINITION:
defun-p (x) = listp $(x) \wedge$ (car (x) = $'\texttt{defun}$)

DEFINITION:
mk-defun $(id, fpl, relation)$ = cons $('\texttt{defun}, [id, fpl, relation])$

DEFINITION:
defun-id $(form)$ = arg* $(form)_0$

DEFINITION:
defun-fpl $(form)$ = arg* $(form)_1$

DEFINITION:
defun-relation $(form)$ = arg* $(form)_2$

DEFINITION:
subprogram-annotation-p (x)
=
assert-p $(x) \vee$ (return-value-p $(x) \vee$ return-relation-p (x))

DEFINITION:
if-p (x) = listp $(x) \wedge$ (car (x) = $'\texttt{if}$)

DEFINITION:
mk-if $(test, then, else)$ = cons $('\texttt{if}, [test, then, else])$

DEFINITION:
if-test $(form)$ = arg* $(form)_0$

DEFINITION:
if-then $(form)$ = arg* $(form)_1$

DEFINITION:
if-else $(form)$ = arg* $(form)_2$

DEFINITION:
set-p (x) = listp $(x) \wedge$ (car (x) = $'\texttt{set}$)

DEFINITION:
mk-set $(id, index, value)$ = cons $('\texttt{set}, [id, index, value])$

DEFINITION:
set-id $(form)$ = arg* $(form)_0$

DEFINITION:
set-index (*form*) = arg* $(form)_1$

DEFINITION:
set-value (*form*) = arg* $(form)_2$

DEFINITION:
get-p (*x*) = listp (*x*) ∧ (car (*x*) = `'get`)

DEFINITION:
mk-get (*id*, *index*) = cons (`'get`, [*id*, *index*])

DEFINITION:
get-id (*form*) = arg* $(form)_0$

DEFINITION:
get-index (*form*) = arg* $(form)_1$

DEFINITION:
assoc-p (*x*) = listp (*x*) ∧ (car (*x*) = `'assoc`)

DEFINITION:
mk-assoc (*x*, *y*) = cons (`'assoc`, [*x*, *y*])

DEFINITION:
assoc-x (*form*) = arg* $(form)_0$

DEFINITION:
assoc-y (*form*) = arg* $(form)_1$

DEFINITION:
lookup-p (*x*) = listp (*x*) ∧ (car (*x*) = `'lookup`)

DEFINITION:
mk-lookup (*x*, *y*) = cons (`'lookup`, [*x*, *y*])

DEFINITION:
lookup-x (*form*) = arg* $(form)_0$

DEFINITION:
lookup-y (*form*) = arg* $(form)_1$

DEFINITION:
in-range-p (*x*) = listp (*x*) ∧ (car (*x*) = `'in-range`)

DEFINITION:
mk-in-range (*x*, *y*) = cons (`'in-range`, [*x*, *y*])

DEFINITION:
in-range-x (*form*) = arg* $(form)_0$

DEFINITION:
in-range-y (*form*) = arg* $(form)_1$

DEFINITION:
not-in-p (*x*) = listp (*x*) ∧ (car (*x*) = `'not-in`)

DEFINITION:
mk-not-in (*x*, *y*) = cons (`'not-in`, [*x*, *y*])

DEFINITION:
not-in-x (*form*) = arg* $(form)_0$

DEFINITION:
not-in-y (*form*) = arg* $(form)_1$

DEFINITION:
$\text{in-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \texttt{'in})$

DEFINITION:
$\text{mk-in}(x, y) = \text{cons}(\texttt{'in}, [x, y])$

DEFINITION:
$\text{in-x}(\mathit{form}) = \text{arg*}(\mathit{form})_{\texttt{0}}$

DEFINITION:
$\text{in-y}(\mathit{form}) = \text{arg*}(\mathit{form})_{\texttt{1}}$

DEFINITION:
$\text{iff-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \texttt{'iff})$

DEFINITION:
$\text{mk-iff}(x, y) = \text{cons}(\texttt{'iff}, [x, y])$

DEFINITION:
$\text{iff-x}(\mathit{form}) = \text{arg*}(\mathit{form})_{\texttt{0}}$

DEFINITION:
$\text{iff-y}(\mathit{form}) = \text{arg*}(\mathit{form})_{\texttt{1}}$

DEFINITION:
$\text{implies-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \texttt{'implies})$

DEFINITION:
$\text{mk-implies}(x, y) = \text{cons}(\texttt{'implies}, [x, y])$

DEFINITION:
$\text{implies-x}(\mathit{form}) = \text{arg*}(\mathit{form})_{\texttt{0}}$

DEFINITION:
$\text{implies-y}(\mathit{form}) = \text{arg*}(\mathit{form})_{\texttt{1}}$

DEFINITION:
$\text{and-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \texttt{'and})$

DEFINITION:
$\text{mk-and}(x, y) = \text{cons}(\texttt{'and}, [x, y])$

DEFINITION:
$\text{and-x}(\mathit{form}) = \text{arg*}(\mathit{form})_{\texttt{0}}$

DEFINITION:
$\text{and-y}(\mathit{form}) = \text{arg*}(\mathit{form})_{\texttt{1}}$

DEFINITION:
$\text{or-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \texttt{'or})$

DEFINITION:
$\text{mk-or}(x, y) = \text{cons}(\texttt{'or}, [x, y])$

DEFINITION:
$\text{or-x}(\mathit{form}) = \text{arg*}(\mathit{form})_{\texttt{0}}$

DEFINITION:
$\text{or-y}(\mathit{form}) = \text{arg*}(\mathit{form})_{\texttt{1}}$

DEFINITION:
$\text{=-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \texttt{'=})$

DEFINITION:
$\text{mk-=}(x, y) = \text{cons}(\texttt{'=}, [x, y])$

DEFINITION:
$=\text{-x}\,(form) = \text{arg*}\,(form)_0$

DEFINITION:
$=\text{-y}\,(form) = \text{arg*}\,(form)_1$

DEFINITION:
$\text{ne-p}\,(x) = \text{listp}\,(x) \wedge (\text{car}\,(x) = \texttt{'ne})$

DEFINITION:
$\text{mk-ne}\,(x, y) = \text{cons}\,(\texttt{'ne}, [x, y])$

DEFINITION:
$\text{ne-x}\,(form) = \text{arg*}\,(form)_0$

DEFINITION:
$\text{ne-y}\,(form) = \text{arg*}\,(form)_1$

DEFINITION:
$\text{lt-p}\,(x) = \text{listp}\,(x) \wedge (\text{car}\,(x) = \texttt{'lt})$

DEFINITION:
$\text{mk-lt}\,(x, y) = \text{cons}\,(\texttt{'lt}, [x, y])$

DEFINITION:
$\text{lt-x}\,(form) = \text{arg*}\,(form)_0$

DEFINITION:
$\text{lt-y}\,(form) = \text{arg*}\,(form)_1$

DEFINITION:
$\text{le-p}\,(x) = \text{listp}\,(x) \wedge (\text{car}\,(x) = \texttt{'le})$

DEFINITION:
$\text{mk-le}\,(x, y) = \text{cons}\,(\texttt{'le}, [x, y])$

DEFINITION:
$\text{le-x}\,(form) = \text{arg*}\,(form)_0$

DEFINITION:
$\text{le-y}\,(form) = \text{arg*}\,(form)_1$

DEFINITION:
$\text{gt-p}\,(x) = \text{listp}\,(x) \wedge (\text{car}\,(x) = \texttt{'gt})$

DEFINITION:
$\text{mk-gt}\,(x, y) = \text{cons}\,(\texttt{'gt}, [x, y])$

DEFINITION:
$\text{gt-x}\,(form) = \text{arg*}\,(form)_0$

DEFINITION:
$\text{gt-y}\,(form) = \text{arg*}\,(form)_1$

DEFINITION:
$\text{ge-p}\,(x) = \text{listp}\,(x) \wedge (\text{car}\,(x) = \texttt{'ge})$

DEFINITION:
$\text{mk-ge}\,(x, y) = \text{cons}\,(\texttt{'ge}, [x, y])$

DEFINITION:
$\text{ge-x}\,(form) = \text{arg*}\,(form)_0$

DEFINITION:
$\text{ge-y}\,(form) = \text{arg*}\,(form)_1$

DEFINITION:
+-p (x) = listp $(x) \wedge$ (car $(x) =$ '+)

DEFINITION:
mk-+ (x, y) = cons ('+, $[x, y]$)

DEFINITION:
+-x $(form)$ = arg* $(form)_0$

DEFINITION:
+-y $(form)$ = arg* $(form)_1$

DEFINITION:
--p (x) = listp $(x) \wedge$ (car $(x) =$ '-)

DEFINITION:
mk-- (x, y) = cons ('-, $[x, y]$)

DEFINITION:
--x $(form)$ = arg* $(form)_0$

DEFINITION:
--y $(form)$ = arg* $(form)_1$

DEFINITION:
-p (x) = listp $(x) \wedge$ (car $(x) =$ ')

DEFINITION:
mk-* (x, y) = cons ('*, $[x, y]$)

DEFINITION:
-x $(form)$ = arg $(form)_0$

DEFINITION:
-y $(form)$ = arg $(form)_1$

DEFINITION:
/-p (x) = listp $(x) \wedge$ (car $(x) =$ '/)

DEFINITION:
mk-/ (x, y) = cons ('/, $[x, y]$)

DEFINITION:
/-x $(form)$ = arg* $(form)_0$

DEFINITION:
/-y $(form)$ = arg* $(form)_1$

DEFINITION:
mod-p (x) = listp $(x) \wedge$ (car $(x) =$ 'mod)

DEFINITION:
mk-mod (x, y) = cons ('mod, $[x, y]$)

DEFINITION:
mod-x $(form)$ = arg* $(form)_0$

DEFINITION:
mod-y $(form)$ = arg* $(form)_1$

DEFINITION:
rem-p (x) = listp $(x) \wedge$ (car $(x) =$ 'rem)

DEFINITION:
mk-rem (x, y) = cons ('rem, $[x, y]$)

DEFINITION:
rem-x (*form*) = arg* (*form*)$_0$

DEFINITION:
rem-y (*form*) = arg* (*form*)$_1$

DEFINITION:
expt-p (*x*) = listp (*x*) ∧ (car (*x*) = `'expt`)

DEFINITION:
mk-expt (*x*, *y*) = cons (`'expt`, [*x*, *y*])

DEFINITION:
expt-x (*form*) = arg* (*form*)$_0$

DEFINITION:
expt-y (*form*) = arg* (*form*)$_1$

DEFINITION:
abs-p (*x*) = listp (*x*) ∧ (car (*x*) = `'abs`)

DEFINITION:
mk-abs (*x*) = cons (`'abs`, [*x*])

DEFINITION:
abs-x (*form*) = arg* (*form*)$_0$

DEFINITION:
not-p (*x*) = listp (*x*) ∧ (car (*x*) = `'not`)

DEFINITION:
mk-not (*x*) = cons (`'not`, [*x*])

DEFINITION:
not-x (*form*) = arg* (*form*)$_0$

DEFINITION:
minus-p (*x*) = listp (*x*) ∧ (car (*x*) = `'minus`)

DEFINITION:
mk-minus (*x*) = cons (`'minus`, [*x*])

DEFINITION:
minus-x (*form*) = arg* (*form*)$_0$

DEFINITION:
append-p (*x*) = listp (*x*) ∧ (car (*x*) = `'append`)

DEFINITION:
mk-append (*x*, *y*) = cons (`'append`, [*x*, *y*])

DEFINITION:
append-x (*form*) = arg* (*form*)$_0$

DEFINITION:
append-y (*form*) = arg* (*form*)$_1$

DEFINITION:
cons-p (*x*) = listp (*x*) ∧ (car (*x*) = `'cons`)

DEFINITION:
mk-cons (*x*) = cons (`'cons`, [*x*])

DEFINITION:
cons-x (*form*) = arg* (*form*)$_0$

DEFINITION:
car-p (x) = listp (x) $\wedge$ (car (x) = `'car`)

DEFINITION:
mk-car (x) = cons (`'car`, [x])

DEFINITION:
car-x ($form$) = arg* ($form$)$_0$

DEFINITION:
cdr-p (x) = listp (x) $\wedge$ (car (x) = `'cdr`)

DEFINITION:
mk-cdr (x) = cons (`'cdr`, [x])

DEFINITION:
cdr-x ($form$) = arg* ($form$)$_0$

DEFINITION:
lexpr-p (x)
=
symbolp (x)
$\vee$ lliteral-p (x)
$\vee$ list-p (x)
$\vee$ in-p (x)
$\vee$ not-in-p (x)
$\vee$ iff-p (x)
$\vee$ implies-p (x)
$\vee$ and-p (x)
$\vee$ or-p (x)
$\vee$ not-p (x)
$\vee$ =-p (x)
$\vee$ range-p (x)
$\vee$ ne-p (x)
$\vee$ lt-p (x)
$\vee$ le-p (x)
$\vee$ gt-p (x)
$\vee$ ge-p (x)
$\vee$ +-p (x)
$\vee$ --p (x)
$\vee$ *-p (x)
$\vee$ /-p (x)
$\vee$ mod-p (x)
$\vee$ rem-p (x)
$\vee$ expt-p (x)
$\vee$ abs-p (x)
$\vee$ minus-p (x)
$\vee$ assoc-p (x)
$\vee$ lookup-p (x)
$\vee$ in-range-p (x)
$\vee$ append-p (x)
$\vee$ cons-p (x)
$\vee$ car-p (x)
$\vee$ cdr-p (x)
$\vee$ if-p (x)
$\vee$ set-p (x)
$\vee$ get-p (x)
$\vee$ instate-p (x)
$\vee$ outstate-p (x))

DEFINITION:
choice-p (x) = range-p $(x) \lor$ (expr-p $(x) \lor$ others-p (x))

DEFINITION:
choices-p (x) = consp $(x) \land$ (car (x) = `'choices`)

DEFINITION:
mk-choices (*args*) = cons (`'choices`, *args*)

DEFINITION:
constant-p (x) = consp $(x) \land$ (car (x) = `'constant`)

DEFINITION:
mk-constant = [`'constant`]

DEFINITION:
variable-p (x) = consp $(x) \land$ (car (x) = `'variable`)

DEFINITION:
mk-variable = [`'variable`]

DEFINITION:
pmode-p (x) = constant-p $(x) \lor$ variable-p (x)

DEFINITION:
fp-spec-p (x) = listp $(x) \land$ (car (x) = `'fp-spec`)

DEFINITION:
mk-fp-spec (*id*, *mode*, *type*) = cons (`'fp-spec`, [*id*, *mode*, *type*])

DEFINITION:
fp-spec-id (*form*) = arg* $(form)_0$

DEFINITION:
fp-spec-mode (*form*) = arg* $(form)_1$

DEFINITION:
fp-spec-type (*form*) = arg* $(form)_2$

DEFINITION:
fpl-p (x) = consp $(x) \land$ (car (x) = `'fpl`)

DEFINITION:
mk-fpl (*args*) = cons (`'fpl`, *args*)

DEFINITION:
number-decl-p (x)
=
listp $(x) \land$ (car (x) = `'number-decl`)

DEFINITION:
mk-number-decl (*id*, *mode*, *body*)
=
cons (`'number-decl`, [*id*, *mode*, *body*])

DEFINITION:
number-decl-id (*form*) = arg* $(form)_0$

DEFINITION:
number-decl-mode (*form*) = arg* $(form)_1$

DEFINITION:
number-decl-body (*form*) = arg* $(form)_2$

DEFINITION:

object-decl-p (x)
=
listp $(x) \wedge (\text{car}(x) = \texttt{'object-decl})$

DEFINITION:
mk-object-decl $(id, mode, type, body)$
=
cons $(\texttt{'object-decl}, [id, mode, type, body])$

DEFINITION:
object-decl-id $(form) = \text{arg*}(form)_{0}$

DEFINITION:
object-decl-mode $(form) = \text{arg*}(form)_{1}$

DEFINITION:
object-decl-type $(form) = \text{arg*}(form)_{2}$

DEFINITION:
object-decl-body $(form) = \text{arg*}(form)_{3}$

DEFINITION:
procedure-p (x)
=
listp $(x) \wedge (\text{car}(x) = \texttt{'procedure})$

DEFINITION:
mk-procedure $(id, params, return, body, spec)$
=
cons $(\texttt{'procedure}, [id, params, return, body, spec])$

DEFINITION:
procedure-id $(form) = \text{arg*}(form)_{0}$

DEFINITION:
procedure-params $(form) = \text{arg*}(form)_{1}$

DEFINITION:
procedure-return $(form) = \text{arg*}(form)_{2}$

DEFINITION:
procedure-body $(form) = \text{arg*}(form)_{3}$

DEFINITION:
procedure-spec $(form) = \text{arg*}(form)_{4}$

DEFINITION:
function-p $(x) = \text{listp}(x) \wedge (\text{car}(x) = \texttt{'function})$

DEFINITION:
mk-function $(id, params, return, body, spec)$
=
cons $(\texttt{'function}, [id, params, return, body, spec])$

DEFINITION:
function-id $(form) = \text{arg*}(form)_{0}$

DEFINITION:
function-params $(form) = \text{arg*}(form)_{1}$

DEFINITION:
function-return $(form) = \text{arg*}(form)_{2}$

DEFINITION:
function-body $(form) = \text{arg*}(form)_{3}$

DEFINITION:
function-spec (*form*) = $\text{arg*}(\textit{form})_4$

DEFINITION:
exception-p (*x*)
=
listp (*x*) ∧ (car (*x*) = `'exception`)

DEFINITION:
mk-exception (*id*) = cons (`'exception`, [*id*])

DEFINITION:
exception-id (*form*) = $\text{arg*}(\textit{form})_0$

DEFINITION:
subprogram-p (*x*) = function-p (*x*) ∨ procedure-p (*x*)

DEFINITION:
use-p (*x*) = consp (*x*) ∧ (car (*x*) = `'use`)

DEFINITION:
mk-use (*args*) = cons (`'use`, *args*)

DEFINITION:
inner-decl-p (*x*)
=
object-decl-p (*x*)
∨ (number-decl-p (*x*) ∨ (assert-p (*x*) ∨ invariant-p (*x*)))

DEFINITION:
inner-decls-p (*x*)
=
consp (*x*) ∧ (car (*x*) = `'inner-decls`)

DEFINITION:
mk-inner-decls (*args*) = cons (`'inner-decls`, *args*)

DEFINITION:
rename-pkg-p (*x*)
=
listp (*x*) ∧ (car (*x*) = `'rename-pkg`)

DEFINITION:
mk-rename-pkg (*new*, *old*) = cons (`'rename-pkg`, [*new*, *old*])

DEFINITION:
rename-pkg-new (*form*) = $\text{arg*}(\textit{form})_0$

DEFINITION:
rename-pkg-old (*form*) = $\text{arg*}(\textit{form})_1$

DEFINITION:
rename-sub-p (*x*)
=
listp (*x*) ∧ (car (*x*) = `'rename-sub`)

DEFINITION:
mk-rename-sub (*new*, *old*) = cons (`'rename-sub`, [*new*, *old*])

DEFINITION:
rename-sub-new (*form*) = $\text{arg*}(\textit{form})_0$

DEFINITION:
rename-sub-old (*form*) = $\text{arg*}(\textit{form})_1$

DEFINITION:
rename-obj-p (x)
$=$
listp $(x) \wedge$ (car $(x) =$ `'rename-obj`)

DEFINITION:
mk-rename-obj (*new*, *type*, *old*)
$=$
cons (`'rename-obj`, [*new*, *type*, *old*])

DEFINITION:
rename-obj-new (*form*) = arg* (*form*)$_0$

DEFINITION:
rename-obj-type (*form*) = arg* (*form*)$_1$

DEFINITION:
rename-obj-old (*form*) = arg* (*form*)$_2$

DEFINITION:
rename-p (x)
$=$
rename-pkg-p $(x) \vee$ (rename-sub-p $(x) \vee$ rename-obj-p (x))

DEFINITION:
decl-p (x)
$=$
inner-decl-p (x)
$\vee$ (subprogram-p (x)
$\vee$ (type-decl-p (x)
$\vee$ (subtype-decl-p (x)
$\vee$ (rename-p (x)
$\vee$ (defun-p $(x) \vee$ (defthm-p $(x) \vee$ defaxiom-p (x)))))))

DEFINITION:
decls-p (x) = consp $(x) \wedge$ (car $(x) =$ `'decls`)

DEFINITION:
mk-decls (*args*) = cons (`'decls`, *args*)

DEFINITION:
raise-p (x) = consp $(x) \wedge$ (car $(x) =$ `'raise`)

DEFINITION:
mk-raise = [`'raise`]

DEFINITION:
null-p (x) = consp $(x) \wedge$ (car $(x) =$ `'null`)

DEFINITION:
mk-null = [`'null`]

DEFINITION:
assign-p (x) = listp $(x) \wedge$ (car $(x) =$ `'assign`)

DEFINITION:
mk-assign (*var*, *value*) = cons (`'assign`, [*var*, *value*])

DEFINITION:
assign-var (*form*) = arg* (*form*)$_0$

DEFINITION:
assign-value (*form*) = arg* (*form*)$_1$

DEFINITION:
proc-call-p (x)
$=$
listp $(x) \land (\text{car}(x) = \texttt{'proc-call})$

DEFINITION:
mk-proc-call $(id, actuals) = \text{cons}(\texttt{'proc-call}, [id, actuals])$

DEFINITION:
proc-call-id $(form) = \text{arg*}(form)_0$

DEFINITION:
proc-call-actuals $(form) = \text{arg*}(form)_1$

DEFINITION:
return-p $(x) = \text{listp}(x) \land (\text{car}(x) = \texttt{'return})$

DEFINITION:
mk-return $(value) = \text{cons}(\texttt{'return}, [value])$

DEFINITION:
return-value $(form) = \text{arg*}(form)_0$

DEFINITION:
exit-p $(x) = \text{consp}(x) \land (\text{car}(x) = \texttt{'exit})$

DEFINITION:
mk-exit $= [\texttt{'exit}]$

DEFINITION:
sl-p $(x) = \text{consp}(x) \land (\text{car}(x) = \texttt{'sl})$

DEFINITION:
mk-sl $(args) = \text{cons}(\texttt{'sl}, args)$

DEFINITION:
while-loop-p (x)
$=$
listp $(x) \land (\text{car}(x) = \texttt{'while-loop})$

DEFINITION:
mk-while-loop $(test, statements)$
$=$
cons $(\texttt{'while-loop}, [test, statements])$

DEFINITION:
while-loop-test $(form) = \text{arg*}(form)_0$

DEFINITION:
while-loop-statements $(form) = \text{arg*}(form)_1$

DEFINITION:
loop-p $(x) = \text{listp}(x) \land (\text{car}(x) = \texttt{'loop})$

DEFINITION:
mk-loop $(statements) = \text{cons}(\texttt{'loop}, [statements])$

DEFINITION:
loop-statements $(form) = \text{arg*}(form)_0$

DEFINITION:
for-loop-p $(x) = \text{listp}(x) \land (\text{car}(x) = \texttt{'for-loop})$

DEFINITION:
mk-for-loop $(var, range, statements)$

=
cons ('`for-loop`, [*var*, *range*, *statements*])

DEFINITION:
for-loop-var (*form*) = arg* $(form)_0$

DEFINITION:
for-loop-range (*form*) = arg* $(form)_1$

DEFINITION:
for-loop-statements (*form*) = arg* $(form)_2$

DEFINITION:
reverse-for-loop-p (*x*)
=
listp (*x*) ∧ (car (*x*) = '`reverse-for-loop`)

DEFINITION:
mk-reverse-for-loop (*var*, *range*, *statements*)
=
cons ('`reverse-for-loop`, [*var*, *range*, *statements*])

DEFINITION:
reverse-for-loop-var (*form*) = arg* $(form)_0$

DEFINITION:
reverse-for-loop-range (*form*) = arg* $(form)_1$

DEFINITION:
reverse-for-loop-statements (*form*) = arg* $(form)_2$

DEFINITION:
loop-stmt-p (*x*)
=
while-loop-p (*x*)
∨ (loop-p (*x*) ∨ (for-loop-p (*x*) ∨ reverse-for-loop-p (*x*)))

DEFINITION:
block-p (*x*) = listp (*x*) ∧ (car (*x*) = '`block`)

DEFINITION:
mk-block (*decls*, *handler*, *body*)
=
cons ('`block`, [*decls*, *handler*, *body*])

DEFINITION:
block-decls (*form*) = arg* $(form)_0$

DEFINITION:
block-handler (*form*) = arg* $(form)_1$

DEFINITION:
block-body (*form*) = arg* $(form)_2$

DEFINITION:
ifarm-p (*x*) = listp (*x*) ∧ (car (*x*) = '`ifarm`)

DEFINITION:
mk-ifarm (*test*, *statements*) = cons ('`ifarm`, [*test*, *statements*])

DEFINITION:
ifarm-test (*form*) = arg* $(form)_0$

DEFINITION:
ifarm-statements (*form*) = arg* $(form)_1$

DEFINITION:
if-stmt-p (x) = consp $(x) \wedge$ (car (x) = `'if-stmt`)

DEFINITION:
mk-if-stmt (*args*) = cons (`'if-stmt`, *args*)

DEFINITION:
casearm-p (x) = listp $(x) \wedge$ (car (x) = `'casearm`)

DEFINITION:
mk-casearm (*test*, *statements*)
=
cons (`'casearm`, [*test*, *statements*])

DEFINITION:
casearm-test (*form*) = arg* $(form)_0$

DEFINITION:
casearm-statements (*form*) = arg* $(form)_1$

DEFINITION:
casearms-p (x) = consp $(x) \wedge$ (car (x) = `'casearms`)

DEFINITION:
mk-casearms (*args*) = cons (`'casearms`, *args*)

DEFINITION:
case-stmt-p (x)
=
listp $(x) \wedge$ (car (x) = `'case-stmt`)

DEFINITION:
mk-case-stmt (*test*, *arms*) = cons (`'case-stmt`, [*test*, *arms*])

DEFINITION:
case-stmt-test (*form*) = arg* $(form)_0$

DEFINITION:
case-stmt-arms (*form*) = arg* $(form)_1$

DEFINITION:
st-compound-p (x)
=
if-stmt-p $(x) \vee$ (case-stmt-p $(x) \vee$ (loop-stmt-p $(x) \vee$ block-p (x)))

DEFINITION:
constrained-st-p (x)
=
listp $(x) \wedge$ (car (x) = `'constrained-st`)

DEFINITION:
mk-constrained-st (*relation*, *stmt*)
=
cons (`'constrained-st`, [*relation*, *stmt*])

DEFINITION:
constrained-st-relation (*form*) = arg* $(form)_0$

DEFINITION:
constrained-st-stmt (*form*) = arg* $(form)_1$

DEFINITION:
st-simple-p (x)
=

$$\begin{array}{l} \quad \text{null-p}\,(x) \\ \vee \quad (\quad \text{assign-p}\,(x) \\ \qquad \vee \quad (\text{proc-call-p}\,(x) \vee (\text{return-p}\,(x) \vee (\text{exit-p}\,(x) \vee \text{raise-p}\,(x))))) \end{array}$$

DEFINITION:
ada-st-p (x) = st-simple-p $(x) \vee$ st-compound-p (x)

DEFINITION:
st-p (x)
=
ada-st-p $(x) \vee$ (constrained-st-p $(x) \vee$ assert-p (x))

DEFINITION:
ids-p (x) = consp $(x) \wedge$ (car (x) = `'ids`)

DEFINITION:
mk-ids (*args*) = cons (`'ids`, *args*)

DEFINITION:
compilation-p (x)
=
consp $(x) \wedge$ (car (x) = `'compilation`)

DEFINITION:
mk-compilation (*args*) = cons (`'compilation`, *args*)

DEFINITION:
comp-unit-p (x)
=
listp $(x) \wedge$ (car (x) = `'comp-unit`)

DEFINITION:
mk-comp-unit (*unit*, *clause*) = cons (`'comp-unit`, [*unit*, *clause*])

DEFINITION:
comp-unit-unit (*form*) = arg* $(\mathit{form})_0$

DEFINITION:
comp-unit-clause (*form*) = arg* $(\mathit{form})_1$

DEFINITION:
context-p (x) = listp $(x) \wedge$ (car (x) = `'context`)

DEFINITION:
mk-context (*with*, *use*) = cons (`'context`, [*with*, *use*])

DEFINITION:
context-with (*form*) = arg* $(\mathit{form})_0$

DEFINITION:
context-use (*form*) = arg* $(\mathit{form})_1$

DEFINITION:
package-p (x) = listp $(x) \wedge$ (car (x) = `'package`)

DEFINITION:
mk-package (*id*, *outer*, *private*, *inner*, *body*)
=
cons (`'package`, [*id*, *outer*, *private*, *inner*, *body*])

DEFINITION:
package-id (*form*) = arg* $(\mathit{form})_0$

DEFINITION:
package-outer (*form*) = arg* $(\mathit{form})_1$

DEFINITION:
package-private (*form*) = arg* (*form*)$_2$

DEFINITION:
package-inner (*form*) = arg* (*form*)$_3$

DEFINITION:
package-body (*form*) = arg* (*form*)$_4$

DEFINITION:
library-unit-p (*x*) = subprogram-p (*x*) ∨ package-p (*x*)

DEFINITION:
compilation-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif comp-unit-p (car (*l*)) **then** compilation-args-p (cdr (*l*))
else nil
fi

DEFINITION:
ids-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif id-p (car (*l*)) **then** ids-args-p (cdr (*l*))
else nil
fi

DEFINITION:
casearms-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif casearm-p (car (*l*)) **then** casearms-args-p (cdr (*l*))
else nil
fi

DEFINITION:
if-stmt-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif ifarm-p (car (*l*)) **then** if-stmt-args-p (cdr (*l*))
else nil
fi

DEFINITION:
sl-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif st-p (car (*l*)) **then** sl-args-p (cdr (*l*))
else nil
fi

DEFINITION:
decls-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif decl-p (car (*l*)) **then** decls-args-p (cdr (*l*))
else nil
fi

DEFINITION:
inner-decls-args-p (*l*)

=
if ¬ consp (*l*) **then t**
elseif inner-decl-p (car (*l*)) **then** inner-decls-args-p (cdr (*l*))
else nil
fi

DEFINITION:
use-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif id-p (car (*l*)) **then** use-args-p (cdr (*l*))
else nil
fi

DEFINITION:
fpl-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif fp-spec-p (car (*l*)) **then** fpl-args-p (cdr (*l*))
else nil
fi

DEFINITION:
choices-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif choice-p (car (*l*)) **then** choices-args-p (cdr (*l*))
else nil
fi

DEFINITION:
apl-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif expr-p (car (*l*)) **then** apl-args-p (cdr (*l*))
else nil
fi

DEFINITION:
designator-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif id-p (car (*l*)) **then** designator-args-p (cdr (*l*))
else nil
fi

DEFINITION:
aggregate-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif aggregate-arm-p (car (*l*)) **then** aggregate-args-p (cdr (*l*))
else nil
fi

DEFINITION:
record-type-args-p (*l*)
=
if ¬ consp (*l*) **then t**
elseif field-spec-p (car (*l*)) **then** record-type-args-p (cdr (*l*))
else nil
fi

DEFINITION:
enumeration-args-p (*l*)
 =
if ¬ consp (*l*) **then t**
 elseif enumeration-literal-p (car (*l*)) **then** enumeration-args-p (cdr (*l*))
 else nil
fi

DEFINITION:
list-args-p (*l*)
 =
if ¬ consp (*l*) **then t**
 elseif expr-p (car (*l*)) **then** list-args-p (cdr (*l*))
 else nil
fi

DEFINITION:
prefix-p-body (*form*)
 =
let *operator* **be** opr (*form*)
  in
case match *operator*:
 case ≅ `'package` **then**
  id-p (package-id (*form*))
  ∧ (((package-outer (*form*) = **nil**)
   ∨ decls-p (package-outer (*form*)))
  ∧ (((package-private (*form*) = **nil**)
   ∨ decls-p (package-private (*form*)))
  ∧ (((package-inner (*form*) = **nil**)
   ∨ decls-p (package-inner (*form*)))
  ∧ ((package-body (*form*) = **nil**)
   ∨ sl-p (package-body (*form*))))))
 case ≅ `'context` **then**
  ((context-with (*form*) = **nil**) ∨ ids-p (context-with (*form*)))
  ∧ ((context-use (*form*) = **nil**)
   ∨ ids-p (context-use (*form*)))
 case ≅ `'comp-unit` **then**
  library-unit-p (comp-unit-unit (*form*))
  ∧ context-p (comp-unit-clause (*form*))
 case ≅ `'compilation` **then** compilation-args-p (arg* (*form*))
 case ≅ `'ids` **then** ids-args-p (arg* (*form*))
 case ≅ `'constrained-st` **then**
  transition-p (constrained-st-relation (*form*))
  ∧ st-compound-p (constrained-st-stmt (*form*))
 case ≅ `'case-stmt` **then**
  expr-p (case-stmt-test (*form*))
  ∧ casearms-p (case-stmt-arms (*form*))
 case ≅ `'casearms` **then** casearms-args-p (arg* (*form*))
 case ≅ `'casearm` **then** expr-p (casearm-test (*form*))
  ∧ sl-p (casearm-statements (*form*))
 case ≅ `'if-stmt` **then** if-stmt-args-p (arg* (*form*))
 case ≅ `'ifarm` **then** expr-p (ifarm-test (*form*))
  ∧ sl-p (ifarm-statements (*form*))
 case ≅ `'block` **then**
  ((block-decls (*form*) = **nil**)
   ∨ inner-decls-p (block-decls (*form*)))
  ∧ (((block-handler (*form*) = **nil**)
   ∨ sl-p (block-handler (*form*)))
  ∧ sl-p (block-body (*form*)))

```
case ≅ 'reverse-for-loop then
        id-p (reverse-for-loop-var (form))
    ∧  (     range-p (reverse-for-loop-range (form))
          ∧  sl-p (reverse-for-loop-statements (form)))
case ≅ 'for-loop then
        id-p (for-loop-var (form))
    ∧  (     range-p (for-loop-range (form))
          ∧  sl-p (for-loop-statements (form)))
case ≅ 'loop then  sl-p (loop-statements (form))
case ≅ 'while-loop then
        expr-p (while-loop-test (form))
    ∧  sl-p (while-loop-statements (form))
case ≅ 'sl then  sl-args-p (arg* (form))
case ≅ 'exit then  t
case ≅ 'return then      (return-value (form) = nil)
                       ∨  expr-p (return-value (form))
case ≅ 'proc-call then      id-p (proc-call-id (form))
                          ∧  apl-p (proc-call-actuals (form))
case ≅ 'assign then      name-p (assign-var (form))
                       ∧  expr-p (assign-value (form))
case ≅ 'null then  t
case ≅ 'raise then  t
case ≅ 'decls then  decls-args-p (arg* (form))
case ≅ 'rename-obj then
        id-p (rename-obj-new (form))
    ∧  (     type-p (rename-obj-type (form))
          ∧  id-p (rename-obj-old (form)))
case ≅ 'rename-sub then
        subprogram-p (rename-sub-new (form))
    ∧  id-p (rename-sub-old (form))
case ≅ 'rename-pkg then      id-p (rename-pkg-new (form))
                           ∧  id-p (rename-pkg-old (form))
case ≅ 'inner-decls then  inner-decls-args-p (arg* (form))
case ≅ 'use then  use-args-p (arg* (form))
case ≅ 'exception then  id-p (exception-id (form))
case ≅ 'function then
        id-p (function-id (form))
    ∧  (     fpl-p (function-params (form))
          ∧  (     id-p (function-return (form))
                ∧  (     (     (function-body (form) = nil)
                                ∨  block-p (function-body (form)))
                      ∧  (     (function-spec (form) = nil)
                                ∨  subprogram-annotation-p (function-spec (form))))))
case ≅ 'procedure then
        id-p (procedure-id (form))
    ∧  (     fpl-p (procedure-params (form))
          ∧  (     (     (procedure-return (form) = nil)
                      ∨  id-p (procedure-return (form)))
                ∧  (     (     (procedure-body (form) = nil)
                                ∨  block-p (procedure-body (form)))
                      ∧  (     (procedure-spec (form) = nil)
                                ∨  subprogram-annotation-p (procedure-spec (form))))))
case ≅ 'object-decl then
        id-p (object-decl-id (form))
    ∧  (     pmode-p (object-decl-mode (form))
          ∧  (     subtype-p (object-decl-type (form))
                ∧  (     (object-decl-body (form) = nil)
                      ∨  expr-p (object-decl-body (form)))))
```

case ≅ ’`number-decl` **then**
  id-p (number-decl-id (*form*))
  ∧ (pmode-p (number-decl-mode (*form*))
    ∧ expr-p (number-decl-body (*form*)))
case ≅ ’`fpl` **then** fpl-args-p (arg* (*form*))
case ≅ ’`fp-spec` **then**
  id-p (fp-spec-id (*form*))
  ∧ (pmode-p (fp-spec-mode (*form*))
    ∧ type-p (fp-spec-type (*form*)))
case ≅ ’`variable` **then** **t**
case ≅ ’`constant` **then** **t**
case ≅ ’`choices` **then** choices-args-p (arg* (*form*))
case ≅ ’`cdr` **then** lexpr-p (cdr-x (*form*))
case ≅ ’`car` **then** lexpr-p (car-x (*form*))
case ≅ ’`cons` **then** lexpr-p (cons-x (*form*))
case ≅ ’`append` **then** lexpr-p (append-x (*form*))
  ∧ lexpr-p (append-y (*form*))
case ≅ ’`minus` **then** lexpr-p (minus-x (*form*))
case ≅ ’`not` **then** lexpr-p (not-x (*form*))
case ≅ ’`abs` **then** lexpr-p (abs-x (*form*))
case ≅ ’`expt` **then** lexpr-p (expt-x (*form*))
  ∧ lexpr-p (expt-y (*form*))
case ≅ ’`rem` **then** lexpr-p (rem-x (*form*))
  ∧ lexpr-p (rem-y (*form*))
case ≅ ’`mod` **then** lexpr-p (mod-x (*form*))
  ∧ lexpr-p (mod-y (*form*))
case ≅ ’`/` **then** lexpr-p (/-x (*form*)) ∧ lexpr-p (/-y (*form*))
case ≅ ’`*` **then** lexpr-p (*-x (*form*)) ∧ lexpr-p (*-y (*form*))
case ≅ ’`-` **then** lexpr-p (--x (*form*)) ∧ lexpr-p (--y (*form*))
case ≅ ’`+` **then** lexpr-p (+-x (*form*)) ∧ lexpr-p (+-y (*form*))
case ≅ ’`ge` **then** lexpr-p (ge-x (*form*))
  ∧ lexpr-p (ge-y (*form*))
case ≅ ’`gt` **then** lexpr-p (gt-x (*form*))
  ∧ lexpr-p (gt-y (*form*))
case ≅ ’`le` **then** lexpr-p (le-x (*form*))
  ∧ lexpr-p (le-y (*form*))
case ≅ ’`lt` **then** lexpr-p (lt-x (*form*))
  ∧ lexpr-p (lt-y (*form*))
case ≅ ’`ne` **then** lexpr-p (ne-x (*form*))
  ∧ lexpr-p (ne-y (*form*))
case ≅ ’`=` **then** lexpr-p (=-x (*form*)) ∧ lexpr-p (=-y (*form*))
case ≅ ’`or` **then** lexpr-p (or-x (*form*))
  ∧ lexpr-p (or-y (*form*))
case ≅ ’`and` **then** lexpr-p (and-x (*form*))
  ∧ lexpr-p (and-y (*form*))
case ≅ ’`implies` **then** lexpr-p (implies-x (*form*))
  ∧ lexpr-p (implies-y (*form*))
case ≅ ’`iff` **then** lexpr-p (iff-x (*form*))
  ∧ lexpr-p (iff-y (*form*))
case ≅ ’`in` **then** lexpr-p (in-x (*form*))
  ∧ lexpr-p (in-y (*form*))
case ≅ ’`not-in` **then** lexpr-p (not-in-x (*form*))
  ∧ lexpr-p (not-in-y (*form*))
case ≅ ’`in-range` **then** lexpr-p (in-range-x (*form*))
  ∧ lexpr-p (in-range-y (*form*))
case ≅ ’`lookup` **then** lexpr-p (lookup-x (*form*))
  ∧ lexpr-p (lookup-y (*form*))
case ≅ ’`assoc` **then** lexpr-p (assoc-x (*form*))

$\wedge$ lexpr-p (assoc-y (*form*))
case ≅ `'get` **then** lexpr-p (get-id (*form*))
$\wedge$ lexpr-p (get-index (*form*))
case ≅ `'set` **then**
lexpr-p (set-id (*form*))
$\wedge$ (lexpr-p (set-index (*form*)) $\wedge$ lexpr-p (set-value (*form*)))
case ≅ `'if` **then**
lexpr-p (if-test (*form*))
$\wedge$ (lexpr-p (if-then (*form*)) $\wedge$ lexpr-p (if-else (*form*)))
case ≅ `'defun` **then**
symbolp (defun-id (*form*))
$\wedge$ (symbolsp (defun-fpl (*form*))
$\wedge$ lexpr-p (defun-relation (*form*)))
case ≅ `'defthm` **then** symbolp (defthm-id (*form*))
$\wedge$ lexpr-p (defthm-relation (*form*))
case ≅ `'defaxiom` **then** symbolp (defaxiom-id (*form*))
$\wedge$ lexpr-p (defaxiom-relation (*form*))
case ≅ `'return-value` **then** lexpr-p (return-value-relation (*form*))
case ≅ `'return-relation` **then**
symbolp (return-relation-var (*form*))
$\wedge$ lexpr-p (return-relation-relation (*form*))
case ≅ `'transition` **then** lexpr-p (transition-relation (*form*))
case ≅ `'invariant` **then** lexpr-p (invariant-relation (*form*))
case ≅ `'assert` **then** lexpr-p (assert-relation (*form*))
case ≅ `'outstate` **then** lexpr-p (outstate-expr (*form*))
case ≅ `'instate` **then** lexpr-p (instate-expr (*form*))
case ≅ `'function-call` **then**
id-p (function-call-id (*form*))
$\wedge$ apl-p (function-call-actuals (*form*))
case ≅ `'op-expr` **then** id-p (op-expr-id (*form*))
$\wedge$ apl-p (op-expr-actuals (*form*))
case ≅ `'apl` **then** apl-args-p (arg* (*form*))
case ≅ `'dot-qual-1` **then**
name-p (dot-qual-1-root (*form*))
$\wedge$ symbolp (dot-qual-1-component (*form*))
case ≅ `'designator` **then** designator-args-p (arg* (*form*))
case ≅ `'selected-component` **then**
expr-p (selected-component-root (*form*))
$\wedge$ symbolp (selected-component-field (*form*))
case ≅ `'apply-1` **then** expr-p (apply-1-root (*form*))
$\wedge$ apl-p (apply-1-args (*form*))
case ≅ `'indexed-component` **then**
expr-p (indexed-component-root (*form*))
$\wedge$ expr-p (indexed-component-index (*form*))
case ≅ `'aggregate` **then** aggregate-args-p (arg* (*form*))
case ≅ `'aggregate-pos` **then** expr-p (aggregate-pos-value (*form*))
case ≅ `'aggregate-choice` **then**
choices-p (aggregate-choice-choice (*form*))
$\wedge$ expr-p (aggregate-choice-value (*form*))
case ≅ `'type-convert` **then**
type-p (type-convert-type (*form*))
$\wedge$ expr-p (type-convert-value (*form*))
case ≅ `'qualified` **then** type-p (qualified-type (*form*))
$\wedge$ expr-p (qualified-value (*form*))
case ≅ `'subtype-decl` **then**
id-p (subtype-decl-id (*form*))
$\wedge$ subtype-p (subtype-decl-decl (*form*))
case ≅ `'type-decl` **then**

```
          id-p (type-decl-id (form))
      ∧ (     (type-decl-decl (form) = nil)
           ∨  type-p (type-decl-decl (form)))
  case ≅ 'array-type then
          type-mark-p (array-type-index (form))
      ∧  type-p (array-type-elements (form))
  case ≅ 'record-type then  record-type-args-p (arg* (form))
  case ≅ 'field-spec then      symbolp (field-spec-id (form))
                            ∧  type-p (field-spec-decl (form))
  case ≅ 'range then       expr-p (range-from (form))
                        ∧  expr-p (range-to (form))
  case ≅ 'attribute then      id-p (attribute-root (form))
                           ∧  id-p (attribute-attr (form))
  case ≅ 'type-mark then
          id-p (type-mark-type (form))
      ∧  constraint-p (type-mark-constraint (form))
  case ≅ 'enumeration then  enumeration-args-p (arg* (form))
  case ≅ 'list then  list-args-p (arg* (form))
  case ≅ 'false then  t
  case ≅ 'true then  t
  case ≅ 'id then
          symbolp (id-root (form))
      ∧  ((id-uid (form) = nil) ∨ integerp (id-uid (form)))
  case ≅ 'unconstrained then  t
  case ≅ 'others then  t
  case ≅ 'error then       listp (error-form (form))
                        ∧  stringp (error-message (form))
  otherwise nil
 endcase
```

THEOREM: strip-cdrs-le *:linear*
acl2-count (strip-cdrs (*x*)) ≤ acl2-count (*x*)

THEOREM: strip-cdrs-lt *:linear*
acl2-count (strip-cdrs (*x*)) < (1 + acl2-count (*x*))

THEOREM: top-literal-p-count *:linear*

```
    acl2-count (strip-cdrs (cdr (form)))
<   (1 + acl2-count (car (form)) + acl2-count (cdr (form)))
```

DEFINITION:
top-literal-p (*form*, *flag*)

```
 =
if flag
 then if ¬ consp (form)  then form = nil
        else     top-literal-p (car (form), nil)
              ∧  top-literal-p (cdr (form), t)
       fi
 elseif booleanp (form)  then t
 elseif integerp (form)  then t
 elseif characterp (form)  then standard-char-p (form)
 elseif array-literal-p (form)  then top-literal-p (range (form), t)
 elseif record-literal-p (form)  then top-literal-p (range (form), t)
 else nil
fi
```

Measure: acl2-count (*form*)

DEFINITION:
top-prefix-p (*form*, *flag*)
 =

if *flag*
then if ¬ consp (*form*) **then** *form* = **nil**
else top-prefix-p (car (*form*), **nil**)
∧ top-prefix-p (cdr (*form*), **t**)
fi
elseif symbolp (*form*) **then t**
elseif integerp (*form*) **then t**
elseif characterp (*form*) ∧ standard-char-p (*form*) **then t**
elseif stringp (*form*) **then t**
elseif lexpr-p (*form*) **then t**
elseif (¬ consp (*form*)) ∨ (¬ listp (arg* (*form*))) **then nil**
elseif car (*form*) ∈ `'(defaxiom defun defthm)` **then t**
elseif top-prefix-p (arg* (*form*), **t**) **then** prefix-p-body (*form*)
else nil
fi
Measure: acl2-count (*form*)

Disable forcing.

EVENT:
PROVE-AVA-PRIMITIVE-TYPE-DEFTHMS

```
( exit null raise
 variable constant false true unconstrained others
 package context comp-unit constrained-st case-stmt
 casearm ifarm block reverse-for-loop for-loop
 loop while-loop return proc-call assign
 rename-obj rename-sub rename-pkg exception function
 procedure object-decl number-decl fp-spec cdr
 car cons append minus not abs expt rem mod
 / * - + ge gt le lt ne = or and
 implies iff in not-in in-range lookup assoc
 get set if defun defthm defaxiom return-value
 return-relation transition invariant assert
 outstate instate function-call op-expr dot-qual-1
 selected-component apply-1 indexed-component
 aggregate-pos aggregate-choice type-convert qualified
 subtype-decl type-decl array-type field-spec range
 attribute type-mark id error)
```

CONSTANT:
ava-primitive-type-repeating =

```
'(compilation ids
  casearms if-stmt sl decls
  inner-decls use fpl choices apl
  designator aggregate record-type
  enumeration list)
```

CONSTANT:
ava-primitive-type-equiv =

```
'(library-unit st
  ada-st st-simple st-compound loop-stmt
  decl rename inner-decl subprogram
  pmode choice lexpr subprogram-annotation
  expr name defining-name aggregate-arm
  type constraint subtype predefined-type
  enumeration-literal lliteral literal
  boolean-literal)
```

DEFINE the theory **ava-primitive-type-fns-2** to be

suffix-fns (*ava-primitive-type-equiv*, `'-p`)
∪ suffix-fns (*ava-primitive-type-repeating*, `'-args-p`).

CONSTANT:
ava-primitive-type-mkfuns =
```
'(mk-package mk-context
  mk-comp-unit mk-compilation mk-ids
  mk-constrained-st mk-case-stmt mk-casearms
  mk-casearm mk-if-stmt mk-ifarm mk-block
  mk-reverse-for-loop mk-for-loop mk-loop
  mk-while-loop mk-sl mk-exit mk-return mk-proc-call
  mk-assign mk-null mk-raise mk-decls mk-rename-obj
  mk-rename-sub mk-rename-pkg mk-inner-decls
  mk-use mk-exception mk-function
  mk-procedure mk-object-decl mk-number-decl mk-fpl
  mk-fp-spec mk-variable mk-constant mk-choices
  mk-cdr mk-car mk-cons mk-append
  mk-minus mk-not mk-abs mk-expt
  mk-rem mk-mod mk-/ mk-* mk--
  mk-+ mk-ge mk-gt mk-le mk-lt
  mk-ne mk-= mk-or mk-and mk-implies
  mk-iff mk-in mk-not-in mk-in-range
  mk-lookup mk-assoc mk-get mk-set
  mk-if mk-defun mk-defthm mk-defaxiom
  mk-return-value mk-return-relation mk-transition
  mk-invariant mk-assert mk-outstate
  mk-instate mk-function-call mk-op-expr
  mk-apl mk-dot-qual-1 mk-designator
  mk-selected-component mk-apply-1
  mk-indexed-component mk-aggregate mk-aggregate-pos
  mk-aggregate-choice mk-type-convert mk-qualified
  mk-subtype-decl mk-type-decl mk-array-type
  mk-record-type mk-field-spec mk-range
  mk-attribute mk-type-mark mk-enumeration
  mk-list mk-false mk-true mk-id
  mk-unconstrained mk-others
  mk-error)
```

CONSTANT:
ava-primitive-type-argfuns =

```
'(error-form error-message
  id-root id-uid type-mark-type
  type-mark-constraint attribute-root attribute-attr
  range-from range-to field-spec-id
  field-spec-decl array-type-index
  array-type-elements type-decl-id type-decl-decl
  subtype-decl-id subtype-decl-decl qualified-type
  qualified-value type-convert-type
  type-convert-value aggregate-choice-choice
  aggregate-choice-value aggregate-pos-value
  indexed-component-root indexed-component-index
  apply-1-root apply-1-args selected-component-root
  selected-component-field dot-qual-1-root
  dot-qual-1-component op-expr-id op-expr-actuals
  function-call-idfunction-call-actuals instate-expr
  outstate-expr assert-relation invariant-relation
  transition-relation return-relation-var
  return-relation-relation return-value-relation
```

```
defaxiom-id defaxiom-relation defthm-id
defthm-relation defun-id defun-fpl
defun-relation if-test if-then if-else
set-id set-index set-value get-id
get-index assoc-x assoc-y lookup-x
lookup-y in-range-x in-range-y not-in-x
not-in-y in-x in-y iff-x iff-y
implies-x implies-y and-x and-y or-x
or-y =-x =-y ne-x ne-y lt-x
lt-y le-x le-y gt-x gt-y ge-x
ge-y +-x +-y --x --y *-x
*-y /-x /-y mod-x mod-y rem-x
rem-y expt-x expt-y abs-x not-x
minus-x append-x append-y cons-x
car-x cdr-x fp-spec-id fp-spec-mode
fp-spec-type number-decl-id number-decl-mode
number-decl-body object-decl-id object-decl-mode
object-decl-type object-decl-body procedure-id
procedure-params procedure-return procedure-body
procedure-spec function-id function-params
function-return function-body function-spec
exception-id rename-pkg-new rename-pkg-old
rename-sub-new rename-sub-old rename-obj-new
rename-obj-type rename-obj-old assign-var
assign-value proc-call-id proc-call-actuals
return-value while-loop-test while-loop-statements
loop-statements for-loop-var for-loop-range
for-loop-statements reverse-for-loop-var
reverse-for-loop-range reverse-for-loop-statements
block-decls block-handler block-body
ifarm-test ifarm-statements casearm-test
casearm-statements case-stmt-test case-stmt-arms
constrained-st-relation constrained-st-stmt
comp-unit-unit comp-unit-clause context-with
context-use package-id package-outer
package-private package-inner
package-body)
```

DEFINE the theory **ava-non-type-syntax-fns** to be **ava-primitive-type-mkfuns** @ **ava-primitive-type-argfuns**.

A.6 Static Semantics Macros

SET CURRENT PACKAGE to be **ACL2**.

INCLUDING the book: *macros*.

INCLUDING the book: *subprefix-openers*.

Some type basics. See also legality-overload.lisp

CONSTANT:
base-integer = `'(id integer 0)`

Enable forcing.

MACRO:
add-variable-binding-to-vs (*entry*, *vs*)

=
```
'(cons ,entry ,vs)
```

@subsection(More on Entries)

DEFINITION:
entry-mode (*x*)
=
if number-decl-p (entry-decl (*x*)) ∨ object-decl-p (entry-decl (*x*))
then entry-decl $(x)_2$
else nil
fi

Does a package have a body? Yes, and its the 5th element.

DEFINITION:
entry-body (*x*)
=
if procedure-p (entry-decl (*x*))
∨ function-p (entry-decl (*x*))
∨ package-p (entry-decl (*x*)) **then** entry-decl $(x)_5$
else nil
fi

DEFINITION:
entry-p (*x*)
=
let *name* **be** entry-name (*x*),
decl **be** entry-decl (*x*),
value **be** entry-value (*x*)
in
id-p (*name*)
∧ (decl-p (*decl*) ∨ subtype-p (*decl*))
∧ (null (*value*) ∨ literal-p (*value*))

inner-decl = object-decl | number-decl | assert | invariant decl = inner-decl | subprogram | type-decl | subtype-decl | rename | defun | defthm | defaxiom

DEFINITION:
decl-id (*d*)
=
if object-decl-p (*d*)
∨ number-decl-p (*d*)
∨ type-decl-p (*d*)
∨ subtype-decl-p (*d*)
∨ subprogram-p (*d*)
∨ rename-pkg-p (*d*)
∨ rename-obj-p (*d*) **then** d_1
elseif rename-sub-p (*d*) **then** rename-sub-new $(d)_1$
else nil
fi

DEFINITION:
decl-kind (*d*)
=
if object-decl-p (*d*) **then** car (object-decl-mode (*d*))
elseif number-decl-p (*d*) **then** `'constant`
elseif type-decl-p (*d*) **then** `'type`
elseif subtype-decl-p (*d*) **then** `'subtype`

elseif procedure-p (*d*) **then** `'procedure`
elseif function-p (*d*) **then** `'function`
elseif rename-pkg-p (*d*) **then** `'rename`
elseif rename-obj-p (*d*) **then** `'rename`
elseif rename-sub-p (*d*) **then** `'rename`
else nil
fi

MODIFY the current theory:

Disable ‘nth’.

During the static semantics check we build an elaboration stack, which is the same thing as a variable stack except the value entry is ignored. The predicate BASIC-ENTRY-P is true of elaboration and value stacks, it does not say anything about what is the value part of this entry.

DEFINITION:
decl-type (*decl*)
=
case on opr (*decl*):
case = `number-decl` **then** **base-integer**
case = `object-decl` **then** object-decl-type (*decl*)
case = `function` **then** function-return (*decl*)
otherwise nil
endcase

DEFINITION:
entry-type (*x*) = decl-type (entry-body (*x*))

DEFINITION:
type-indication-p (*x*)
=
predefined-type-p (*x*) ∨ id-p (*x*) ∨ type-mark-p (*x*)

DEFINITION:
basic-entry-p (*entry*)
=
entry-p (*entry*)
∧ subtype-p (entry-decl (*entry*))
∧ literal-p (entry-value (*entry*))

@subsection(Ava literals, values and types)

Moved definition of ADD-VARIABLE-BINDING-TO-VS from ava-dynamic. A. Flatau 4-May-1994

We have decided @i[not] to require all our values carry type information. Why overspecify? In the case of integers, we know what we want our operations to do, so why burden ourselves with useless type information.

For us, the "value" part of an object may be either an actual "raw" value or an expression. Or an apple, for that matter.

A elaboration-stack is a list of basic-entry-p’s.

DEFINITION:
elaboration-stack-p (*alist*)
=
if consp (*alist*)

then basic-entry-p (car (*alist*)) ∧ elaboration-stack-p (cdr (*alist*))
else *alist* = **nil**
fi

Later we may make requirements on the environment that say, for example, that every variable is assigned a value. For now we will make only trivial "type-theoretic" requirements.

MACRO:
lookup (*x*, *env*)
=
```
‘(lookup2 ,x (es ,env))
```

DEFINITION:
lookup3 (*x*, *ea*)
=
if ¬ consp (*ea*) **then nil**
elseif *x* = entry-name (car (*ea*)) **then** car (*ea*)
else lookup3 (*x*, cdr (*ea*))
fi

DEFINITION:
lookup2 (*x*, *es*)
=
if ¬ consp (*es*) **then nil**
elseif lookup3 (*x*, car (*es*)) **then nil**
else lookup2 (*x*, cdr (*es*))
fi

DEFINITION:
variable-lookup (*x*, *env*)
=
let *entry* **be** lookup (*x*, *env*)
in
if subtype-p (entry-decl (*entry*)) **then** *entry*
else nil
fi

DEFINITION:
proc-lookup (*x*, *env*)
=
let *entry* **be** lookup (*x*, *env*)
in
if procedure-p (entry-decl (*entry*)) **then** entry-decl (*entry*)
else nil
fi

DEFINITION:
proc-definedp (*x*, *env*)
=
let *entry* **be** lookup (*x*, *env*)
in
procedure-p (entry-decl (*entry*))

DEFINITION:
func-lookup (*x*, *env*)
=
let *entry* **be** lookup (*x*, *env*)
in
if function-p (entry-decl (*entry*)) **then** entry-decl (*entry*)
else nil

fi

DEFINITION:
func-definedp (*x*, *env*)
 =
let *entry* **be** lookup (*x*, *env*)
in
function-p (entry-decl (*entry*))

THEOREM: basic-entry-p-fact1
basic-entry-p (*entry*) $\rightarrow$ entry-p (*entry*)

THEOREM: basic-entry-p-fact2
basic-entry-p (*entry*) $\rightarrow$ id-p (entry-name (*entry*))

THEOREM: basic-entry-p-fact3
basic-entry-p (*entry*) $\rightarrow$ subtype-p (entry-decl (*entry*))

THEOREM: basic-entry-p-fact4
basic-entry-p (*entry*) $\rightarrow$ literal-p (entry-value (*entry*))

(defthm basic-entry-p-facts (and (implies (basic-entry-p entry) (entry-p entry)) (implies (basic-entry-p entry) (id-p (entry-name entry))) (implies (basic-entry-p entry) (member (entry-kind entry) '(constant variable))) (implies (basic-entry-p entry) (type-p (entry-body entry)))) :Hints (("Goal" :in-theory (disable expr-p type-p id-p))))

MODIFY the current theory:

Disable 'basic-entry-p'.

A.7 Predefined Packages

SET CURRENT PACKAGE to be **ACL2**.

CONSTANT:
standard =

```
'(package
  (id standard 0)
  (decls
   (type-decl (id boolean 0) (enumeration (false) (true)))
   (subtype-decl (id integer 0)
                 (type-mark (id base-integer 0)
                            (range (id ava_min_int 0) (id ava_max_int 0))))
   (subtype-decl (id natural 0)
                 (type-mark (id integer 0) (range 0 (id ava_max_int 0))))
   (subtype-decl (id positive 0)
                 (type-mark (id integer 0) (range 1 (id ava_max_int 0))))
   (function (id abs 0)
             (fpl (fp-spec (id left 0) (constant) (id integer 0)))
             (id integer 0)
             nil nil)
   (function (id rem 0)
             (fpl (fp-spec (id left 0) (constant) (id integer 0))
                  (fp-spec (id right 0) (constant) (id integer 0)))
             (id integer 0)
             nil nil)
   (function (id mod 0)
             (fpl (fp-spec (id left 0) (constant) (id integer 0))
                  (fp-spec (id right 0) (constant) (id integer 0)))
             (id integer 0)
             nil nil)
   (type-decl (id character 0)
```

```
              (enumeration            -- nul ... us
              #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\!
              #\" #\# #\$ #\% #\& #\'
              #\( #\) #\* #\+ #\, #\-
              #\. #\/ #\0 #\1 #\2 #\3
              #\4 #\5 #\6 #\7 #\8 #\9
              #\: #\; #\< #\= #\> #\?
              #\@ #\a #\b #\c #\d #\e
              #\f #\g #\h #\i #\j #\k
              #\l #\m #\n #\o #\p #\q
              #\r #\s #\t #\y #\v #\w
              #\x #\y #\z #\[ #\\ #\]
              #\^ #\_ #\‘ #\a #\b #\c
              #\d #\e #\f #\g #\h #\i
              #\j #\k #\l #\m #\n #\o
              #\p #\q #\r #\s #\t #\u
              #\v #\w #\x #\y #\z #\{
              #\| #\} #\~ #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  #\  #\
              #\  #\  #\  #\  ))

   (package (id ascii 0) nil nil nil nil)
   (type-decl (id string 0)
              (array-type (type-mark (id positive 0) (unconstrained))
                          (id character 0)))
   (wide_character)
   (float)
   (wide_string)
   (exception (id program_error 0))))
```

CONSTANT:
**ada* =*

```
'(package (id ada 0)
  (decls
   (package (id ava-io 0)
            (decls
             (type-decl (id file_type 0) nil)
             (type-decl (id file_mode 0) nil)

             (procedure (id close 0)
```

```
                     (fpl (fp-spec (id file 0) (constant) (id file_mode 0)))
                     nil nil nil)

          (function (id mode 0)
                    (fpl (fp-spec (id file 0) (constant) (id file_mode 0)))
                    (id file_mode 0)
                    nil nil)
          (function (id is_open 0)
                    (fpl (fp-spec (id file 0) (constant) (id file_mode 0)))
                    (id boolean 0)
                    nil nil)
          (function (id end_of_file 0)
                    (fpl (fp-spec (id file 0) (constant) (id file_mode 0)))
                    (id boolean 0)
                    nil nil)

          (object-decl (id standard_output 0) (constant) (id file_type 0) nil)
          (object-decl (id standard_input 0) (constant) (id file_type 0) nil)

          (object-decl (id eol 0) (constant) (character) nil)

          (procedure (id get 0)
                     (fpl (fp-spec (id file 0) (constant) (id file_mode 0))
                          (fp-spec (id item 0) (variable) (id character 0)))
                     nil nil nil)
          (procedure (id put 0 )
                     (fpl (fp-spec (id file 0) (constant) (id file_mode 0))
                          (fp-spec (id item 0) (constant) (id character 0)))
                     nil nil nil)

          (procedure (id get_line 0)
                     (fpl (fp-spec (id file 0) (constant) (id file_mode 0))
                          (fp-spec (id item 0) (variable) (id string 0)))
                     nil nil nil)

          (procedure (id put_line 0)
                     (fpl (fp-spec (id file 0) (constant) (id file_mode 0))
                          (fp-spec (id item 0) (constant) (id string 0)))
                     nil nil nil)))

(package (id io_exceptions 0)
 (decls (exception (id status_error 0))
        (exception (id mode_error 0))
        (exception (id name_error 0))
        (exception (id use_error 0))
        (exception (id device_error 0))
        (exception (id end_error 0))
        (exception (id data_error 0))
        (exception (id layout_error 0))))

(package (id system 0)
 (decls (type-decl (id name 0) nil)
        (object-decl (id ava_system_name 0) (constant) (id name 0))
        (object-decl (id ava_min_int 0) (constant) (id integer 0))
        (object-decl (id ava_max_int 0) (constant) (id integer 0))))))
```

CONSTANT:
annex-a = [**ada**, **standard**]

References

[Smith 95] M. K. Smith.
AVA 95 Reference Manual.
Technical Report 114, Computational Logic, Inc., September, 1995.
Derived from ISO/IEC 8652:1995(E).

Index

Table of Contents

List of Figures

List of Tables