12. Predicates and Specification Expressions

This chapter describes predicates in JML and JML's extensions to Java's expressions. It also describes store references, which are similar to specification expressions, but are used to describe locations instead of values. Details are found in the sections below.

12.1 Predicates

A *predicate* The following gives the syntax of predicates, which are simply *spec-expression*s that must have a boolean value. See section 12.2 Specification Expressions, for the syntax of specification expressions.

predicate ::= *spec-expression*

12.2 Specification Expressions

The following gives the syntax of specification expressions in JML. See section 12.3 Expressions, for the syntax of *expression*.

spec-expression-list ::= *spec-expression*
                  [, *spec-expression*] …
spec-expression ::= *expression*

Within a *spec-expression*, one cannot use any of the operators (such as `++`, `--`, and the assignment operators) that would necessarily cause side effects. In addition, one can use extensions that are specific to JML, in particular the JML primary expressions.

12.3 Expressions

The JML syntax for expressions extends the Java syntax with several operators and primitives.

The precedence of operators in JML expressions is similar to that in Java The precedence levels are given in the following table, where the parentheses, quantified expressions, `[]`, `.`, and method calls on the first three lines all have the highest precedence, and for the rest, only the operators on the same line have the same precedence.

highest	`new () \forall \exists \max \min`
	`\num_of \product \sum` *informal-description*
	`[] .` and method calls

```
          unary + and - ~ ! (typecast)
          * / %
          + (binary) - (binary)
          << >> >>>
          < <= > >= <: instanceof
          == !=
          &
          ^
          |
          &&
          ||
          ==> <==
          <==> <=!=>
          ?:
lowest    = *= /= %= += -= <<= >>= >>>= &= ^= |=
```

The following is the syntax of Java expressions, with JML additions. The additions are the operators `==>`, `<==`, `<==>`, `<=!=>`, and `<:`, and the syntax found under the nonterminals *jml-primary* (see section 12.4 JML Primary Expressions) and *set-comprehension* (see section 12.5 Set Comprehensions). The JML additions to the Java syntax can only be used in assertions and other annotations. Furthermore, within assertions, one cannot use any of the operators (such as `++`, `--`, and the assignment operators) that would necessarily cause side effects.

expression-list ::= *expression* [`,` *expression*] ...
expression ::= *assignment-expr*
assignment-expr ::= *conditional-expr*
    [*assignment-op assignment-expr*]
assignment-op ::= `=` | `+=` | `-=` | `*=` | `/=` | `%=` | `>>=`
    | `>>>=` | `<<=` | `&=` | `` ` ``|=' | `^=`
conditional-expr ::= *equivalence-expr*
    [`?` *conditional-expr* `:` *conditional-expr*]
equivalence-expr ::= *implies-expr*
    [*equivalence-op implies-expr*] ...
equivalence-op ::= `<==>` | `<=!=>`
implies-expr ::= *logical-or-expr*
    [`==>` *implies-non-backward-expr*]
  | *logical-or-expr* `<==` *logical-or-expr*
    [`<==` *logical-or-expr*] ...
implies-non-backward-expr ::= *logical-or-expr*
    [`==>` *implies-non-backward-expr*]
logical-or-expr ::= *logical-and-expr* [`` ` ``||' *logical-and-expr*] ...
logical-and-expr ::= *inclusive-or-expr* [`&&` *inclusive-or-expr*] ...
inclusive-or-expr ::= *exclusive-or-expr* [`` ` ``|' *exclusive-or-expr*] ...
exclusive-or-expr ::= *and-expr* [`^` *and-expr*] ...
and-expr ::= *equality-expr* [`&` *equality-expr*] ...
equality-expr ::= *relational-expr* [`==` *relational-expr*] ...
  | *relational-expr* [`!=` *relational-expr*] ...
relational-expr ::= *shift-expr* `<` *shift-expr*
  | *shift-expr* `>` *shift-expr*
  | *shift-expr* `<=` *shift-expr*
  | *shift-expr* `>=` *shift-expr*
  | *shift-expr* `<:` *shift-expr*
  | *shift-expr* [`instanceof` *type-spec*]
shift-expr ::= *additive-expr* [*shift-op additive-expr*] ...
shift-op ::= `<<` | `>>` | `>>>`
additive-expr ::= *mult-expr* [*additive-op mult-expr*] ...
additive-op ::= `+` | `-`

mult-expr ::= *unary-expr* [*mult-op* *unary-expr*] ...
mult-op ::= `*` | `/` | `%`
unary-expr ::= `(` *type-spec* `)` *unary-expr*
  | `++` *unary-expr*
  | `--` *unary-expr*
  | `+` *unary-expr*
  | `-` *unary-expr*
  | *unary-expr-not-plus-minus*
unary-expr-not-plus-minus ::= `~` *unary-expr*
  | `!` *unary-expr*
  | `(` *built-in-type* `)` *unary-expr*
  | `(` *reference-type* `)` *unary-expr-not-plus-minus*
  | *postfix-expr*
postfix-expr ::= *primary-expr* [*primary-suffix*] ... [`++`]
  | *primary-expr* [*primary-suffix*] ... [`--`]
  | *built-in-type* [`` `[' `]' ``] ... `. class`
primary-suffix ::= `.` *ident*
  | `. this`
  | `. class`
  | `.` *new-expr*
  | `. super` `(` [*expression-list*] `)`
  | `(` [*expression-list*] `)`
  | `` `[' `` *expression* `` `]' ``
  | [`` `[' `]' ``] ... `. class`
primary-expr ::= *ident* | *new-expr*
  | *constant* | `super` | `true`
  | `false` | `this` | `null`
  | `(` *expression* `)`
  | *jml-primary*
built-in-type ::= `void` | `boolean` | `byte`
  | `char` | `short` | `int`
  | `long` | `float` | `double`
constant ::= *java-literal*
new-expr ::= `new` *type* *new-suffix*
new-suffix ::= `(` [*expression-list*] `)` [*class-block*]
  | *array-decl* [*array-initializer*]
  | *set-comprehension*
array-decl ::= *dim-exprs* [*dims*]
dim-exprs ::= `` `[' `` *expression* `` `]' `` [`` `[' `` *expression* `` `]' ``] ...
array-initializer ::= `{` [*initializer* [`,` *initializer*] ... [`,`]] `}`
initializer ::= *expression*
  | *array-initializer*

[[[Need to have semantics of the new things explained here.]]]

12.4 JML Primary Expressions

The following is the syntax of *jml-primary*.

jml-primary ::= *result-expression*
  | *old-expression*
  | *not-assigned-expression*
  | *not-modified-expression*
  | *only-accessed-expression*
  | *only-assigned-expression*
  | *only-called-expression*

| *only-captured-expression*
| *fresh-expression*
| *reach-expression*
| *duration-expression*
| *space-expression*
| *working-space-expression*
| *nonnullelements-expression*
| *informal-description*
| *typeof-expression*
| *elemtype-expression*
| *type-expression*
| *lockset-expression*
| *max-expression*
| *is-initialized-expression*
| *invariant-for-expression*
| *lblneg-expression*
| *lblpos-expression*
| *spec-quantified-expr*

All of the JML keywords that can be used in expressions which would otherwise start with an alphabetic character start with a backslash (\), so that they cannot clash with the program's variable names.

The new expressions that JML introduces are described below. Several of the descriptions below quote, without attribution, descriptions from [Leavens-Baker-Ruby06].

12.4.1 `\result`

The syntax of a *result-expression* is as follows.

result-expression ::= `\result`

The primary `\result` can only be used in `ensures`, `duration`, and `workingspace` clauses of a non-void method. Its value is the value returned by the method. Its type is the return type of the method; hence it is a type error to use `\result` in a void method or in a constructor.

12.4.2 `\old` and `\pre`

An *old-expression* has the following syntax. See section 12.2 Specification Expressions, for the syntax of *spec-expression*.

old-expression ::= `\old` `(` *spec-expression* [`,` *ident*] `)`
    | `\pre` `(` *spec-expression* `)`

An expression of the form `\old(`*Expr*`)` refers to the value that the expression *Expr* had in the pre-state of a method.

JML uses Java's reference semantics, hence the pre-state value of an expression whose type is a reference type is simply the reference; it is *not* a clone of the object the reference points to. For example, suppose in the pre-state that `v` is field that holds a reference to a `HashMap`; concretely, suppose that the location stored in `v` is 0x952ab340. Then the expression `\old(v)` denotes the pre-state value of `v`, which is the same reference, i.e., it is the address 0x952ab340. Note that `\old(v)` is not a reference to a copy of the `HashMap` stored at that location, but simply a copy of the location's address (the reference), which is the value of `v`. If the fields of the object at that location have changed in the post-state, then changes to those fields will be visible through `\old(v)`; for example, `\old(v).size()` will be the same as `v.size()`. To write a post-condition that refers to `v`'s size in the pre-state, one should instead write `\old(v.size())`. Indeed as a general rule, it is always safest to use `\old()` only around expressions whose type is a value type or a type with immutable values, such as `String`.

Expressions of this form may be used in both normal and exceptional postconditions (see section 9. Method Specifications, for more about such `ensures` and `signals` clauses), in history constraints, in duration and working space clauses, and also in assertions that appear in the bodies of methods (see section 13. Statements and Annotation Statements, for more about `assert` and `assume` statements, loop invariants, and variant functions).

However, we recommend that inside the bodies of methods, one of the two other forms of *old-expression* (see below) be used instead. The reason for this is that the reader may wonder whether `\old(Expr)` in the body of a method means the pre-state value of *Expr* (which it does) or the value of *Expr* before some previous statement (which it does not).

An expression of the form `\pre(`*Expr*`)` also refers to the value that the expression *Expr* had in the pre-state of a method. Expressions of this form may only be used in assertions that appear in the bodies of methods (i.e., in `assert` and `assume` statements, and in loop invariants and variant functions). That is, such expressions may not be used in specification cases, and hence may not appear in normal or exceptional postconditions, in history constraints, or in duration and working space clauses.

An expression of the form `\old(`*Expr*`,` *Label*`)` refers to the value that the expression *Expr* had when control last reached the statement label *Label*. That is, it refers to the value of the expression just before control last reached the statement the label is attached to. Expressions of this form may only be used when the *Label* has been declared in a surrounding context. Thus such expressions may be used in `assert` and `assume` statements, and in loop invariants and variant functions, where such a label has been declared. They may also be used in specification cases that occur in model program *spec-statement* (see section 15.6 Specification Statements) and in the *refining-statements* (see section 13.4.3 Refining Statements).

In an expression of the form `\old(`*Expr*`,` *Label*`)`, *Label* must be a label defined in the current method. The type of `\old(`*Expr*`)`, `\old(`*Expr*`,` *Label*`)`, or `\pre(`*Expr*`)`, is simply the type of *Expr*.

12.4.3 `\not_assigned`

The syntax of a *not-assigned-expression* is as follows. See section 12.7 Store Refs, for the syntax of *store-ref-list*.

not-assigned-expression ::= `\not_assigned` `(` *store-ref-list* `)`

The JML operator `\not_assigned` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. It asserts that the locations in the data group (see section 10. Data Groups) named by the argument were not assigned to during the execution of the method being specified (or all methods to which a history constraint applies). For example, `\not_assigned(xval,yval)` says that the locations in the data groups named by `xval` and `yval` were not assigned during the method's execution.

A predicate such as `\not_assigned(x.f)` refers to the entire data group named by `x.f` not just to the location `x.f` itself. This allows one to specify absence of even temporary side-effects in various cases of a method. See section 12.4.4 `\not_modified`, for ways to specify that just the value of a given field was not changed, which allows temporary side effects.

The `\not_assigned` operator can be applied to both concrete and model or ghost fields. When applied to a model field, the meaning is that all (concrete) locations in that model field's data group were not assigned. [[[A real example would help here.]]]

The type of a `\not_assigned` expression is `boolean`.

12.4.4 `\not_modified`

The syntax of a *not-modified-expression* is as follows. See section 12.7 Store Refs, for the syntax of *store-ref-list*.

not-modified-expression ::= `\not_modified` `(` *store-ref-list* `)`

The JML operator `\not_modified` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. It asserts that the values of the named fields are the same in the post-state as in the pre-state; for example, `\not_modified(xval,yval)` says that the fields `xval` and `yval` have the same value in the pre- and post-states (in the sense of the `equals` method for their types).

A predicate such as `\not_modified(x.f)` refers to the location named by `x.f`, not to the entire data group of `x.f`. This allows one to specify benevolent side-effects, as one can name `x.f` (or a data group in which it participates) in an assignable clause, but use `\not_modified(x.f)` in the postcondition. See section 12.4.3 `\not_assigned`, for ways to specify that no assignments were made to any location in a data group, disallowing temporary side effects.

The `\not_modified` operator can be applied to both concrete and model or ghost fields. When applied to a model field, the meaning is that only the value of the model field is unchanged (in the sense of its type's equals operation); concrete fields involved in its representation may have changed. [[[A real example would help here.]]]

The type of a `\not_modified` expression is `boolean`.

12.4.5 `\only_accessed`

The syntax of an *only-accessed-expression* is as follows. See section 12.7 Store Refs, for the syntax of *store-ref-list*.

only-accessed-expression ::= `\only_accessed` `(` *store-ref-list* `)`

The JML operator `\only_accessed` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. Used in a method's postcondition (perhaps implicitly in a history constraint), it asserts that the method's execution only reads from a subset of the data groups named by the given fields. For example, `\only_accessed(xval,yval)` says that no fields, outside of the data groups of `xval` and `yval` were read by the method. This includes both direct reads in the body of the method, and reads during calls that were made by the method (and methods those methods called, etc.).

A predicate such as `\only_accessed(x.f)` refers to the entire data group named by `x.f` not just to the location `x.f` itself.

The `\only_accessed` operator can be applied to both concrete and model or ghost fields. When applied to a model field, the meaning is that the (concrete) locations in that model field's data group are permitted to be accessed during the method's execution.

The type of an `\only_accessed` expression is `boolean`.

12.4.6 `\only_assigned`

The syntax of an *only-assigned-expression* is as follows. See section 12.7 Store Refs, for the syntax of *store-ref-list*.

only-assigned-expression ::= `\only_assigned` `(` *store-ref-list* `)`

The JML operator `\only_assigned` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. Used in a method's postcondition (perhaps implicitly in a history constraint), it asserts that the method's execution only assigned to a subset of the data groups named by the given fields. For example, `\only_assigned(xval,yval)` says that no fields, outside of the data groups of `xval` and `yval` were assigned by the method. This includes both direct assignments in the body of the method, and assignments during calls that were made by the method (and methods those methods called, etc.).

A predicate such as `\only_assigned(x.f)` refers to the entire data group named by `x.f` not just to the location `x.f` itself.

The `\only_assigned` operator can be applied to both concrete and model or ghost fields. When applied to a model field, the meaning is that the (concrete) locations in that model field's data group are permitted to be assigned during the method's execution.

The type of an `\only_assigned` expression is `boolean`.

12.4.7 `\only_called`

The syntax of an *only-called-expression* is as follows. See section 8.3 Constraints, for the syntax of *method-name-list*.

only-called-expression ::= `\only_called` `(` *method-name-list* `)`

The JML operator `\only_called` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. Used in a method's postcondition (perhaps implicitly in a history constraint), it asserts that the method's execution only called from a subset of methods given in the *method-name-list*. For example, `\only_called(p,q)` says that methods, apart from `p` and `q`, were called during this method's execution.

The type of an `\only_called` expression is `boolean`.

12.4.8 `\only_captured`

The syntax of an *only-captured-expression* is as follows. See section 12.7 Store Refs, for the syntax of *store-ref-list*.

only-captured-expression ::= `\only_captured ( ` *store-ref-list* `)`

The JML operator `\only_captured` can be used in both normal and exceptional preconditions (i.e., in `ensures` and `signals` clauses), and in history constraints. Used in a method's postcondition (perhaps implicitly in a history constraint), it asserts that the method's execution only captured references from a subset of the data groups named by the given fields. For example, `\only_captured(xv,yv)` says that no references, outside of the data groups of `xv` and `yv` were captured by the method.

A reference is *captured* when it is stored into a field (as opposed to a local variable). Typically a method captures a formal parameter (or a reference stored in a static field) by assigning it to a field in the method's receiver (the `this` object), a field in some object (or to an array element), or to a static field.

A predicate such as `\only_captured(x.f)` refers to the references stored in the entire data group named by `x.f` in the pre-state, not just to those stored in the location `x.f` itself. However, since the references being captured are usually found in formal parameters, the complications of data groups can usually be ignored.

The `\only_captured` operator can be applied to both concrete and model or ghost fields. When applied to a model field, the meaning is that the (concrete) locations in that model field's data group are permitted to be captured during the method's execution.

The type of an `\only_captured` expression is `boolean`.

12.4.9 `\fresh`

The syntax of a *fresh-expression* is as follows. See section 12.2 Specification Expressions, for the syntax of *spec-expression-list*.

fresh-expression ::= `\fresh ( ` *spec-expression-list* `)`

The operator `\fresh` asserts that objects were freshly allocated; for example, `\fresh(x,y)` asserts that `x` and `y` are not null and that the objects bound to these identifiers were not allocated in the pre-state. The arguments to `\fresh` can have any reference type, and the type of the overall expression is `boolean`.

Note that it is wrong to use `\fresh(this)` in the specification of a constructor, because Java's `new` operator allocates storage for the object; the constructor's job is just to initialize that storage.

12.4.10 `\reach`

The syntax of a *reach-expression* is as follows. See section 12.2 Specification Expressions, for the syntax of *spec-expression*.

reach-expression ::= `\reach ( ` *spec-expression* `)`

The `\reach` expression allows one to refer to the set of objects reachable from some particular object. The syntax `\reach(`*x*`)` denotes the smallest `JMLObjectSet` containing the object denoted by *x*, if any, and all objects accessible through all fields of objects in this set. That is, if *x* is `null`, then this set is empty otherwise it contains *x*, all objects accessible through all fields of *x*, all objects accessible through all fields of these objects, and so on, recursively. If *x* denotes a model field (or data group), then `\reach(`*x*`)` denotes the smallest `JMLObjectSet` containing the objects reachable from *x* or reachable from the objects referenced by fields in that data group.

12.4.11 `\duration`

The syntax of a *duration-expression* is as follows. See section 12.3 Expressions, for the syntax of *expression*.

duration-expression ::= `\duration` (*expression*)

`\duration`, which describes the specified maximum number of virtual machine cycle times needed to execute the method call or explicit constructor invocation expression that is its argument; e.g., `\duration(myStack.push(o))` is the maximum number of virtual machine cycles needed to execute the call `myStack.push(o)`, according to the contract of the static type of `myStack`'s type's `push` method, when passed argument `o`. Note that the expression used as an argument to `\duration` should be thought of as quoted, in the sense that it is not to be executed; thus the method or constructor called need not be free of side effects. Note that the argument to `\duration` is an *expression* instead of just the name of a method, because different method calls, i.e., those with different parameters, can take different amounts of time [Krone-Ogden-Sitaraman03].

The argument expression passed to `\duration` must be a method call or explicit constructor invocation expression; the type of a `\duration` expression is `long`.

For a given Java Virtual Machine, a *virtual machine cycle* is defined to be the minimum of the maximum over all Java Virtual Machine instructions, *i*, of the length of time needed to execute instruction *i*.

12.4.12 `\space`

The syntax of a *space-expression* is as follows. See section 12.2 Specification Expressions, for the syntax of *spec-expression*. [[[Shouldn't this take an expression instead of a spec-expression? - DRC]]]

space-expression ::= `\space` (*spec-expression*)

`\space`, which describes the amount of heap space, in bytes, allocated to the object referred to by its argument [Krone-Ogden-Sitaraman03]; e.g., `\space(myStack)` is number of bytes in the heap used by `myStack`, not including the objects it contains. The type of the *spec-expression* that is the argument must be a reference type, and the result type of a `\space` expression is `long`.

12.4.13 `\working_space`

working-space-expression ::= `\working_space` (*expression*)

`\working_space`, which describes the maximum specified amount of heap space, in bytes, used by the method call or explicit constructor invocation expression that is its argument; e.g., `\working_space(myStack.push(o))` is the maximum number of bytes needed on the heap to execute the call `myStack.push(o)`, according to the contract of the static type of `myStack`'s type's `push` method, when passed argument `o`. Note that the expression used as an argument to `\working_space` should be thought of as quoted, in the sense that it is not to be executed; thus the method or constructor called need not be free of side effects. The detailed arguments are needed in the specification of the call because different method calls, i.e., those with different parameters, can use take different amounts of space [Krone-Ogden-Sitaraman03]. The argument expression must be a method call or explicit constructor invocation expression; the result type of a `\working_space` expression is `long`.

12.4.14 `\nonnullelements`

The syntax of a *nonnullelements-expression* is as follows. See section 12.2 Specification Expressions, for the syntax of *spec-expression*.

nonnullelements-expression ::= `\nonnullelements` `(` *spec-expression* `)`

The operator `\nonnullelements` can be used to assert that an array and its elements are all non-null. For example, `\nonnullelements(myArray)`, is equivalent to [Leino-Nelson-Saxe00]

```
myArray != null &&
(\forall int i; 0 <= i && i < myArray.length;
                myArray[i] != null)
```

12.4.15 Informal Predicates

An *informal-description* is some text enclosed in `(*` and `*)`. See section 4.6 Tokens, for details of its syntax. It is used as an escape form formality.

An informal description used as a predicate has type boolean. Hence the text in an informal description should describe a condition, for example `(* the value of x is displayed *)`.

The value of an informal description is only known to the user, not to any JML tools, so it is never executable. Informal descriptions should thus be avoided when possible, but can be used to avoid formalizing everything when doing so would be too expensive.

12.4.16 `\typeof`

The syntax of a *typeof-expression* is as follows. See section 12.2 Specification Expressions, for the syntax of *spec-expression*.

typeof-expression ::= `\typeof` `(` *spec-expression* `)`

The operator `\typeof` returns the most-specific dynamic type of an expression's value [Leino-Nelson-Saxe00]. The meaning of `\typeof(`*E*`)` is unspecified if *E* is null. If *E* has a static type that is a reference type, then `\typeof(`*E*`)` means the same thing as *E*`.getClass()`. For example, if `c` is a variable of static type `Collection` that holds an object of class `HashSet`, then `\typeof(c)` is `HashSet.class`, which is the same thing as `\type(HashSet)`. If *E* has a static type that is not a reference type, then `\typeof(`*E*`)` means the instance of `java.lang.Class` that represents its static type. For example, `\typeof(true)` is `Boolean.TYPE`, which is the same as `\type(boolean)`. Thus an expression of the form `\typeof(`*E*`)` has type `\TYPE`, which JML considers to be the same as `java.lang.Class`.

12.4.17 `\elemtype`

The syntax of a *elemtype-expression* is as follows.

elemtype-expression ::= `\elemtype` `(` *spec-expression* `)`

The `\elemtype` operator returns the most-specific static type shared by all elements of its array argument [Leino-Nelson-Saxe00]. For example, `\elemtype(\type(int[]))` is `\type(int)`. The argument to `\elemtype` must be an expression of type `\TYPE`, which JML considers to be the same as `java.lang.Class`, and its result also has type `\TYPE` (see section 7.1.2.2 Type-Specs). If the argument is not an array type, then the result is `null`. For example, `\elemtype(\type(int))` and `\elemtype(\type(Object))` are both `null`.

12.4.18 `\type`

The syntax of a *type-expression* is as follows. See section 7.1.2.2 Type-Specs, for the syntax of *type*.

type-expression ::= `\type (` *type* `)`

The operator `\type` can be used to introduce literals of type `\TYPE` in expressions. An expression of the form `\type(T)`, where `T` is a type name, has the type `\TYPE`. Since in JML `\TYPE` is the same as `java.lang.Class`, an expression of the form `\type(`*T*`)` means the same thing as *T*`.class`, if *T* is a reference type. If *T* is a primitive type, then `\type(T)` is equivalent to the value of the `TYPE` field of the corresponding reference type. Thus `\type(boolean)` equals `Boolean.TYPE`.

For example, in

```
    \typeof(myObj) <: \type(PlusAccount)
```

the use of `\type(PlusAccount)` is used to introduce the type `PlusAccount` into this expression context.

12.4.19 `\lockset`

The syntax of a *lockset-expression* is as follows.

lockset-expression ::= `\lockset`

The `\lockset` primitive denotes the set of locks held by the current thread. It is of type `JMLObjectSet`. (This is an adaptation from ESC/Java [Leino-etal00] [Leino-Nelson-Saxe00] for dealing with threads.)

12.4.20 `\max`

The syntax of a *max-expression* is as follows. See section 12.2 Specification Expressions, for the syntax of *spec-expression*.

max-expression ::= `\max (` *spec-expression* `)`

The `\max` operator returns the "largest" (as defined by `<`) of a set of lock objects, given a lock set as an argument. The result is of type Object. (This is an adaptation from ESC/Java [Leino-etal00] [Leino-Nelson-Saxe00] for dealing with threads.)

If you are looking to take the maximum of several integers, use the max quantifier (see section 12.4.24.2 Generalized Quantifiers).

12.4.21 `\is_initialized`

The syntax of the *is-initialized-expression* is as follows. See section 7.1.2.2 Type-Specs, for the syntax of *reference-type*

is-initialized-expression ::= `\is_initialized (` *reference-type* `)`

The `\is_initialized` operator returns true just when its *reference-type* argument is a class that has finished its static initialization. It is of type `boolean`.

12.4.22 `\invariant_for`

invariant-for-expression ::= `\invariant_for` `(` *spec-expression* `)`

The `\invariant_for` operator returns true just when its argument satisfies the invariant of its static type; for example, `\invariant_for((MyClass)o)` is true when `o` satisfies the invariant of `MyClass`. The entire `\invariant_for` expression is of type `boolean`.

12.4.23 `\lblneg` and `\lblpos`

The syntax of the two kinds of labeled expressions is as follows. See section 12.2 Specification Expressions, for the syntax of *spec-expression*.

lblneg-expression ::= `(` `\lblneg` *ident spec-expression* `)`
lblpos-expression ::= `(` `\lblpos` *ident spec-expression* `)`

Parenthesized expressions that start with `\lblneg` and `\lblpos` can be used to attach labels to expressions [Leino-Nelson-Saxe00]; these labels might be printed in various messages by support tools, for example, to identify an assertion that failed. Such an expression has a *label* and a *body*; for example, in

```
(\lblneg indexInBounds 0 <= index && index < length)
```

the label is `indexInBounds` and the body is the expression `0 <= index && index < length`. The value of a labeled expression is the value of its body, hence its type is the type of its body. The idea is that if this expression is used in an assertion and its value is `false` (e.g., when doing run-time checking of assertions), then a warning will be printed that includes the label `indexInBounds`. The form using `\lblpos` has a similar syntax, but should be used for warnings when the value of the enclosed expression is `true`.

12.4.24 Quantified Expressions

spec-quantified-expr ::= `(` *quantifier quantified-var-decls* `;`
    [[*predicate*] `;`]
    spec-expression `)`
quantifier ::= `\forall` | `\exists`
    | `\max` | `\min`
    | `\num_of` | `\product` | `\sum`
quantified-var-decls ::= [*bound-var-modifiers*] *type-spec quantified-var-declarator*
    [`,` *quantified-var-declarator*] ...
quantified-var-declarator ::= *ident* [*dims*]
spec-variable-declarators ::= *spec-variable-declarator*
    [`,` *spec-variable-declarator*] ...
spec-variable-declarator ::= *ident* [*dims*]
    [`=` *spec-initializer*]
spec-array-initializer ::= `{` [*spec-initializer*
    [`,` *spec-initializer*] ... [`,`]] `}`
spec-initializer ::= *spec-expression*
    | *spec-array-initializer*

Note that each quantified expression includes a set of parentheses; these parentheses cannot be omitted. The first part of a quantified expression is the *quantifier*, which determines the operation to be performed. Every quantifier starts with a backslash (`\`). Following the quantifier are *quantified-var-decls*, which declare *bound variables* whose scope is the *spec-quantified-expr*. The bound variables may not conflict with existing local variables, but may hide static and instance fields. The optional predicate between the two semicolons is the *range predicate*; a quantifier ranges over all possible values of its bound variables that satisfy the range predicate (for a discussion of the ranges of values for reference types, see section 12.4.24.6 Quantifying over Reference Types). If the range predicate is omitted, it defaults to `true`. The final *spec-expression* is called the *body* of the quantifier.

We discuss the various kinds of quantified expressions below.

12.4.24.1 Universal and Existential Quantifiers

The quantifiers `\forall` and `\exists`, are universal and existential quantifiers (respectively). For example,

```
(\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])
```

says that the values `a[0]` ... `a[9]` are sorted.

The body of a universal or existential quantifier must be of type `boolean`. The type of a universal or existential quantified expression as a whole is `boolean`. When the range predicate is not satisfiable, the value of a `\forall` expression is `true` and the value of an `\exists` expression is `false`. For example:

```
(\forall int i; 0 < i && i < 0; 0 < i) == true
(\exists int i; 0 < i && i < 0; 0 < i) == false
```

12.4.24.2 Generalized Quantifiers

The quantifiers `\max`, `\min`, `\product`, and `\sum`, are generalized quantifiers that return the maximum, minimum, product, or sum of the values of the expressions given, where the variables satisfy the given range. The expression in the body must be of a built-in numeric type, such as `int` or `double`; the type of the quantified expression as a whole is the type of its body. For example, the following equations are all true (see chapter 3 of [Cohen90]):

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
(\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4
(\max int i; 0 <= i && i < 5; i) == 4
(\min int i; 0 <= i && i < 5; i-1) == -1
```

For computing the value of a sum or product, Java's arithmetic is used. [[[This would depend on the arithmetic mode in force - DRC]]]The meaning thus depends on the type of the expression. For example, in Java, floating point numbers use the IEEE 754 standard, and thus when an overflow occurs, the appropriate positive or negative infinity is returned. However, Java integers wrap on overflow. Consider the following examples.

```
(\product float f; 1.0e30f < f && f < 1.0e38f; f)
  == Float.POSITIVE_INFINITY

(\sum int i; i == Integer.MAX_VALUE || i == 1; i)
  == Integer.MAX_VALUE + 1
  == Integer.MIN_VALUE
```

When the range predicate is not satisfiable, the sum is 0 and the product is 1; for example:

```
(\sum int i; false; i) == 0
```

```
    (\product double d; false; d*d) == 1.0
```

When the range predicate is not satisfiable for `\max` the result is the smallest number with the type of the expression in the body; for floating point numbers, negative infinity is used. Similarly, when the range predicate is not satisfiable for `\min`, the result is the largest number with the type of the expression in the body. [[[Or should this be undefined - DRC]]]

12.4.24.3 Numerical Quantifier

The numerical quantifier, `\num_of`, returns the number of values for its variables for which the range and the expression in its body are true. The body must have type `boolean`, and the entire quantified expression has type `long`. The meaning of this quantifier is defined by the following equation (see p. 57 of [Cohen90]).

```
    (\num_of T x; R(x); P(x)) == (\sum T x; R(x) && P(x); 1L)
```

12.4.24.4 Executability of Quantified Expressions

When are universal or existential quantifiers executable for purposes of runtime assertion checking? If the type of the quantified variable is `boolean`, then it is always executable. Otherwise a *spec-quantified-expr* is only executable if the form of the expression matches a pattern that the runtime assertion checker understands. This varies by tool implementation, but you can expect that the runtime assertion checker understands patterns where the range predicate gives a finite range for an ordinal primitive value type (such as `int`) or where the range predicate requires the quantified variable to be drawn from some set. Examples include the following. [[[Make these examples be real examples in the samples directory]]]

```
        (\forall int x; 0 <= x && x < somelimit; ...)
        (\forall Object x; someSet.has(x); ...)
```

You should get warnings from the `jmlc` tool when assertions are not executable, but you have to use the `-w2` flag to see them.

If a *spec-quantified-expr*, *QE*, is executable, then a tool executing it should only evaluate any range expression in *QE* once per execution of *QE*. Since the value of such a range expression cannot change, this evaluation strategy will not change the value of *QE*, but it will save time to only evaluate the range expression once for each evaluation of *QE*.

12.4.24.5 Modifiers for Bound Variables

bound-var-modifiers ::= `non_null` | `nullable`

Logical variables can be bound in

- quantified expressions (see section 12.4.24 Quantified Expressions),
- set comprehension expressions (see section 12.5 Set Comprehensions),
- forall clauses of method contracts (see section 9.9.1.1 Forall Variable Declarations), or
- old clauses of method contracts (see section 9.9.1.2 Old Variable Declarations).

Note that in JML, `non_null` and `nullable` are not reserved words, hence such identifiers can be used as type names. In order to quantify over the elements of a type named `non_null` or `nullable` is necessary to provide an explicit nullity modifier. For example,

```
    (\forall non_null non_null nn; ...)
```

where the first `non_null` is one of the *bound-var-modifiers* and the second is the type `non_null`.

12.4.24.6 Quantifying over Reference Types

The range of values for a quantified variable that is declared to be of a reference type:

- Does not include `null` unless the bound variable is declared `nullable` (see section G.2.1 Non-null by Default).
- May include references to objects that are not constructed by the program; one should use a range predicate to eliminate such cases if they are not desired.

12.5 Set Comprehensions

The syntax of a *set-comprehension* expression is as follows.

set-comprehension ::= { [*bound-var-modifiers*] *type-spec*
    quantified-var-declarator `|'
    postfix-expr `&&` *predicate* `}`

The set comprehension notation can be used to succinctly define sets. The meaning of a *new-expr* (see section 12.3 Expressions) with a *set-comprehension* suffix, such as `new` *ST* `{` *T x* `|` *s*`.has(`*x*`)` `&&` *P*`(`*x*`)` `}` is to form a subset, of type *ST*, of the set *s*, which contains just those elements *x* that are both in *s* and for which *P*`(`*x*`)` is true.

For example, the following is the `JMLObjectSet` that is the subset of non-`null` `Integer` objects found in the set `myIntSet` whose values are between 0 and 10, inclusive.

```
  new JMLObjectSet {Integer i | myIntSet.has(i) &&
      i != null && 0 <= i.intValue() && i.intValue() <= 10 }
```

The syntax of JML limits set comprehensions so that the *postfix-expr* following the vertical bar (`|`) is always a method invocation with the bound variable declared in the *quantified-var-declarator* as its parameter; the method may be either the `has` method of an `org.jmlspecs.models.JMLObjectSet` or `org.jmlspecs.models.JMLValueSet`, or the `contains` method of a `java.util.Collection`. This restriction is used to avoid Russell's paradox [Whitehead-Russell25]. The bound variable, whose scope is the *set-comprehension*, may not conflict with existing local variables, but may hide static and instance fields. The bound variable type is used to restrict the objects that become part of the resulting set; if the set called in the *postfix-expr* contains objects that are not assignable to the bound variable, they are not contained in the resulting set comprehension. Thus, the following two set comprehension expressions (given an existing `Collection s`) result in identical sets:

```
  new JMLObjectSet {Integer i | s.contains(i) && 0 < i.intValue() }
  new JMLObjectSet {Object i | s.contains(i) && i instanceof Integer &&
      0 < ((Integer) i).intValue() }
```

In practice, one starts either from some relevant set at hand or from the sets found in `JMLObjectSet` and `JMLValueSet` containing the objects of primitive types. The type of a set comprehension is the type named following `new`, which must be `JMLObjectSet` or `JMLValueSet`. The bound variable type must be compatible with the set comprehension type; in particular, the bound variable type must be a subtype of `org.jmlspecs.models.JMLType` if the set comprehension type is `JMLValueSet`.

12.6 JML Operators

In this section we describe the various new operators that JML adds to Java expressions. The following can all be used in *spec-expressions*.

12.6.1 Subtype operator

The relational operator `<:` compares two reference types and returns true when the type on the left is a subtype of the type on the right [Leino-Nelson-Saxe00]. Although the notation might suggest otherwise, this operator is also reflexive; a type will compare as `<:` with itself. In an expression of the form *E1* `<:` *E2*, both *E1* and *E2* must have type `\TYPE`; since in JML `\TYPE` is the same as `java.lang.Class` the expression *E1* `<:` *E2* means the same thing as the expression *E2*`.isAssignableFrom(`*E1*`)`. As a result, primitive types are not subtypes of `java.lang.Object`, nor of each other, though they are of themselves; so, for example, `Integer.TYPE <: Integer.TYPE` is true.

12.6.2 Equivalence and Inequivalence Operators

The operators `<==>` and `<=!=>` work only on boolean-subexpressions and have the same meaning as `==` and `!=`, respectively. However, they have very low precedence, and so are useful at the top-level of a *spec-expression*. Unlike `==` and `!=`, the operators `<==>` and `<=!=>` are also associative and symmetric.

The notation `<==>` can be read "if and only if". It has the same meaning for Boolean values as `==`, but has a lower precedence. Therefore, the expression "`\result <==> size == 0`" means the same thing as "`\result == (size == 0)`".

The notation `<=!=>` can be read "is not equivalent to". It has the same meaning for Boolean values as `!=`, but has a lower precedence. Therefore, the expression "`\result <=!=> size == 0`" means the same thing as "`\result != (size == 0)`".

The expressions on either side of these operators must be of type `boolean`, and the type of the result is also `boolean`.

12.6.3 Forward and Reverse Implication Operators

The operators `==>` and `<==` work only on boolean-subexpressions. They compute forward and reverse implications, respectively.

For example, the formula `raining ==> getsWet` is true if either `raining` is false or `getsWet` is true. The formula `getsWet <== raining` means the same thing. The `==>` operator associates to the right, but the `<==` operator associates to the left. The expressions on either side of these operators must be of type `boolean`, and the type of the result is also `boolean`.

These two operators are evaluated in short-circuit fashion, left to right. Thus, in `a ==> b`, if `a` is false, then the expression is true and `b` is not evaluated. Similarly, in `a <== b`, if `a` is true, the expression is true and `b` is not evaluated. In other words, `a ==> b` is equivalent to `!a || b` and `a <== b` is equivalent to `a || !b`.

Because of this short-circuit evaluation, `a ==> b` is not quite equivalent to `b <== a`. For example, `x != null ==> x.a > 0` will be true if `x` is `null`, but `x.a>0 <== x != null` would be undefined (or throw a NullPointerException) if `x` is `null`.

12.6.4 Lockset Ordering

JML uses `<#` and `<#=` to test order of locks. (The previously-used operators `<` and `<=` were deprecated because their use conflicts with the Java comparisons defined for those operators when autoboxing is available.)

Using `synchronized` statements, Java programs can establish monitor locks to permit only one thread at a time to execute given sections of code. Any object can be used as a lock. In order for ESC/Java [Leino-Nelson-Saxe00] to reason about the possibility of deadlocks among threads, a partial order must be statically declared on lock objects, with "larger" objects being objects whose locks should be acquired later. ESC/Java suggests the use of *axiom-clause*s to declare this partial order.

The `<#` and `<#=` operators test this partial order in assertions. When used in this way, the subexpressions to either side of `<#` or `<#=` must be reference types, and the result is of type boolean.

12.7 Store Refs

The syntax related to the *store-ref* production is used in several places, in particular in assignable clauses (see section 9.9.9 Assignable Clauses).

store-ref-list ::= *store-ref-keyword* | *store-ref* [`,` *store-ref*] ...
store-ref ::= *store-ref-expression*
        | *informal-description*
store-ref-expression ::= *store-ref-name* [*store-ref-name-suffix*] ...
store-ref-name ::= *ident* | `super` | `this`
store-ref-name-suffix ::= `.` *ident* | `. this` | `` `[' `` *spec-array-ref-expr* `` `]' `` | `. *`
spec-array-ref-expr ::= *spec-expression*
        | *spec-expression* `..` *spec-expression*
        | `*`
store-ref-keyword ::= `\nothing` | `\everything` | `\not_specified`

A *store-ref* denotes a set of locations. These sets can be specified using data groups (see section 10. Data Groups), and if this is done then the set of locations denoted by a *store-ref* is the union of all the sets of locations in the specified set of data groups.

The set of locations denoted by a *store-ref-list* of the form *store-ref*`,` [*store-ref*] ... is the union of all the sets of locations denoted by each *store-ref* in the list.

The *store-ref* `\nothing` denotes the empty set of locations. The form `\everything` denotes the set of all locations in the program. The form `\not_specified` denotes a unspecified set of locations, whose usage is determined by a particular tool.

When *SR* denotes a set of locations contain objects, then *SR*.*f*, where *f* is an *ident*, denotes the union of the data groups of each field named *f* in each object in the denotation of *SR*.

The meaning of a *store-ref-expression* of the form *SR*`.*` depends on the denotation of the *stor-ref-name* *SR*. When *SR* denotes a set of locations of objects, then *SR*`.*` denotes the union of the data groups of all visible instance fields of *SR*'s (static) type. On the other hand, if *SR* names a class or interface, then *SR*`.*` denotes the union of the data groups of all visible static fields of the named class or interface.

Similarly, when *SR* denotes a set of locations containing arrays, then *SR*`[*]` denotes the union of all data groups of all elements in all the arrays denoted by *SR*. Also, when *SR* denotes a set of locations containing arrays, then *SR*`[`*L*`..`*H*`]` denotes the union of all data groups of all elements in the arrays denoted by *SR* whose indexes are between *L* and *H* inclusive. In the case where *SR* denotes a set of locations containing arrays, then *SR*`[`*J*`]` denotes the union of all data groups of those arrays at the index denoted by the *spec-expression* *J*.

[<<] [>>]   [Top] [Contents] [Index] [?]

This document was generated by *U-leavens-nd\leavens* on *May, 31 2013* using *texi2html*