Chapter 8
From Specification to Proof Obligations

Daniel Grahl and Mattias Ulbrich

Specification with the Java Modeling Language (JML) has been introduced by example in the previous chapter without giving formal definitions of the meaning of JML specifications. Unfortunately, the JML reference manual [Leavens et al., 2013] does not provide a *formal* semantics, but informal descriptions, often stated in operational terms of the Java language. This is a serious shortcoming since the primary use case of JML is formal specification. Some formal representations from within the JML community have been suggested before [Jacobs and Poll, 2001, Engel, 2005, Darvas and Müller, 2007, Bruns, 2009], but none of them prevailed. Furthermore, over the years several extensions or dialects to JML have emerged (e.g., the extension with dynamic frames by Weiß [2011] that is used in KeY).

In the present chapter, we provide a denotational formal semantics to JML by translating expressions and contracts to formulas in JavaDL. It thus links Chapter 3 on JavaDL with Chapter 7 on JML. We assume that the reader is familiar with JavaDL and with the basic syntax of JML, and that he or she has an intuitive sense of their semantics. This chapter gives a comprehensive definition of the semantics of JML, more specifically of the dialect of the specification language used in the KeY system. It is marked where the semantics presented in this chapter refines or deviates from that given in the reference manual.

The chapter is divided into three sections: In Section 8.1, we define a translation from JML expressions to JavaDL terms and formulas. In Section 8.2, we introduce JavaDL contracts to which JML specifications are translated. Finally, in Section 8.3, we give proof obligations for contracts, i.e., we explain which formulas need to be proven for a given program to be correct w.r.t. its contract. We distinguish three kinds of proof obligations: (1) functional correctness proof obligations (Section 8.3.1), (2) dependency proof obligations (Section 8.3.2), and (3) well-definedness proof obligations (Section 8.3.3). The full definition comprises many similar cases which we present exemplarily. A full account of the JML semantics can be found in Appendix A.

We focus on *local* correctness of a single method implementation. This means that we only cover the provider's side of a contract in the design by contract framework. Chapter 9 on modular specification and verification goes beyond this and defines correctness of whole programs. Calculus rules that use contracts can be found there. Chapter 13 on information flow introduces yet another extension to JML, which is based on the semantics described in this chapter and generalizes the concept of dependency proof obligations from Section 8.3.2.

8.1 Formal Semantics of JML Expressions

We start by giving semantics to JML expressions by providing a translation to JavaDL terms or formulas (see Chapter 3). While JML has been designed to be intuitively understandable, in particular expression syntax being similar to first-order logic, a formal account is not always straightforward. On a closer look, the fact that JML semantics relies on the Java program that is being specified renders the whole issue more complex than expected. In particular, JML contains several implicit assumptions that are not contained per se in JavaDL.

A note on notation: We use typewriter font for terminal (program) syntax elements—such as local variables or operators—and math font for nonterminal JML expressions.

We fix a Java program Prg with JML annotations. Let JTypes_{Prg} denote the set of JML types, including the reference types that are defined in Prg. Let JExp_{Prg} denote the set of well-formed JML expressions w.r.t. Prg according to syntax and typing rules defined in the JML reference manual [Leavens et al., 2013].

JML expressions are statically typed such that for every operator the argument and result types can be inferred at compile-time. Thus the translation of overloaded operators distinguishes cases by types, e.g., the binary JML operator `&` is translated to either logical conjunction (if the operands are of type Boolean) or an appropriate bit vector operation (if the operands are of an integer type).

In JML, the concept 'formula' does not exist; its place is rather taken by Boolean expressions. JavaDL, on the other hand, distinguishes between Boolean terms and formulas. For a translation from JML to JavaDL we need to distinguish both cases and define the translation from JML expressions as a mapping to either terms or formulas in JavaDL.

Definition 8.1. Let $\mathscr{T}$ be a JavaDL type hierarchy for a Java program Prg. The translation function $\lfloor\cdot\rfloor : \mathrm{JExp}_{Prg} \cup \mathrm{JTypes}_{Prg} \to \mathrm{Trm}_{Any} \cup \mathrm{DLFml} \cup \mathscr{T}$ maps JML expressions and types to terms, formulas, or types in JavaDL. It is defined in Tables A.6–A.11 in Appendix A.2. Whether the result is a formula or a term depends on the context; if necessary, a Boolean term x can be converted to a formula $x \doteq TRUE$.

Logical symbols

The translated JavaDL terms may contain the predefined function symbols of the signature Σ_J (see Figure 2.4). In addition, they may contain the program variables and function symbols (i) `self` for the reference to the current receiver object (i.e., the equivalent to `this` in Java), (ii) `heap` for the current heap, (iii) `heap`^{pre} for the prestate heap, (iv) `exc` for an exception to be raised (and not caught) by the program, (v) `res` for the result of a method, as well as (vi) any local variables, parameters, and field identifiers defined by the Java program. The symbols `heap`^{pre}, `exc`, and `res` only appear in postconditions.

In the remainder of this section, we discuss the translation for representative examples.

8.1.1 Types in JML

The type system of JML comprises the type system of Java and extends it with the specification-only types `\bigint` (mathematical integers), `\real` (real numbers), and `\TYPE` (the type of all types). The JML dialect used in KeY further introduces the types `\seq` (finite sequences), and `\locset` (location sets).[1] All Java/JML types that occur in the program under inspection (both primitive types and defined classes and interfaces) have a direct counterpart in JavaDL.

In JML, the different integer data types (`byte`, `short`, `char`, `int`, `long`, and `\bigint`) refer to different ranges of mathematical integers while there is only one type *int* in JavaDL—representing the mathematical integers $\mathbb{Z}$. Note that these types are not subtypes of each other, but rather *retrenchments* [Schlager, 2002]. All JML integer types are translated to the same domain *int* in JavaDL. To account for the different value ranges, there exist restriction predicates $inInt(x)$, $inByte(x)$, etc. for each integral type except `\bigint`. The semantics of this type restriction depends on the choice of integer semantics in JavaDL; the resulting formulas will be different (see Section 5.4): For instance $I_{\mathrm{Java}}(inInt) = [-2^{31}, 2^{31}-1]$ while $I_{\mathrm{math}}(inInt) = \mathbb{Z}$.

All mentioned specification-only types (`\real`, `\bigint`, `\TYPE`, `\seq` and `\locsec`) are *primitive* data types, put in Java lingo. Note that the JML modeling classes, like `JMLObjectSet`, are not primitive but reference types. This chapter does not cover some of the available primitive data types: floating-point types (`float` and `double`) as well as the JML types `\real` and `\TYPE` are currently not supported in KeY. There is no translation into JavaDL for them. The interested reader can refer to [Bruns, 2009]. In contrast to the JML reference manual [Leavens et al., 2013], we do not allow array types such as `\bigint[]`. Unlike all other array types, they are *not* subtypes of `java.lang.Object` and therefore expose some semantical irregularities.

Reference types are mapped to their equally named Java correspondence. A distinction is made concerning whether or not the value `null` is included in the translation. By default, bound variables in JML do not include `null` as possible value. This default restriction is motivated by the observation that `null` dereference is a common

[1] For the underlying theories of finite sequences and location sets, see Sections 2.4 and 5.2, respectively. Usage of the `\seq` and `\locset` data types in JML specifications will be discussed in Sections 8.1.3 and 9.3.2, respectively.

source of programming errors. Chalin and Rioux [2005] reported on the observation that the majority of type references used in declarations in a Java program are designed to hold only values different from `null`. It is therefore natural to assume this constraint as the implicitly assumed default and have all situations in which `null` is an admissible value be annotated explicitly. This shortens specifications and enhances safety by making contracts stricter by default. Nonnull types are built into the Eiffel language [Meyer, 1989]. For Java, there are dedicated static checkers for nonnull annotated types (see [Chalin et al., 2008]).

We understand the JML type **`non_null`** $T \in \mathrm{JTypes}$ (which excludes the `null` reference) as a subtype to the respective unrestricted type **`nullable`** T (i.e., the actual Java reference type). In JavaDL we represent both types as the same type T and encode the nonnullness as a constraint. We define a family of formulas $\mathrm{inRange}_T(x)$ that represents the restrictions on term x to JML type $T \in \mathrm{JTypes}$. This type restriction formula is used anywhere in JavaDL formulas where range restriction is required, e.g., in preconditions or in quantifier ranges. Note that the symbol $\mathrm{inRange}()$ does not actually occur in formulas, but is merely an abbreviation used in this book.

For a reference type T that is not an array type, $\mathrm{inRange}_T(x)$ is defined as the following formula:

$$x.created \doteq TRUE \land x \not\doteq \texttt{null}$$

For reference array types, e.g., `Object[]`, there is a further restriction that all array entries $x[i]$ are different from `null` as well—even in depth in case of multi-dimensional arrays.[2] To encapsulate this 'deep nonnull,' we use the recursively defined predicate $nonNull(h,x,d)$ which means that in heap h reference x is not `null` for dimension d, see Section 8.2.1.2 for the formal definition.

Similar to the definition for reference types, the type **`\locset`** is restricted to location sets in which all members of the set belong to allocated objects. There is no JML expression denoting an unallocated object, and there is no way of constructing an expression that denotes a set that contains unallocated locations. In Section 9.3.4, we will encounter an example for why it is useful that dynamic frames in JML never contain unallocated locations.

8.1.2 Translating JML Expressions to JavaDL

This subsection explains the translation of selected, relevant operators. A comprehensive list can be found in Appendix A.2.

[2] The translation does not include that the entries must be created. It is an implicit axiom in JavaDL that all referenced objects are created. This is captured in the semantics of the *wellFormed* predicate, see Definition 3.5.

8.1.2.1 Boolean Logical Expressions

Boolean expressions are translated in a straightforward manner. For most JML operators, there is simply an alternative JavaDL syntax. E.g., all three expressions A `!=` B, A `<=!=>` B, and A`^`B are translated to $\neg(\lfloor A \rfloor \leftrightarrow \lfloor B \rfloor)$. For quantified expressions, we add a type restriction to the range, as discussed above. E.g., `(\forall int x;` A`;` B`)` is translated to $\forall\ \mathit{int}\ x; (\mathit{inInt}(x) \wedge \lfloor A \rfloor \rightarrow \lfloor B \rfloor)$, where we assume the bound variable `x` in JML to be identified with the logical variable x. According to the JML reference manual [Leavens et al., 2013, Sect. 12.4.24.6], the range of quantification over reference types "may include references to objects that are not constructed by the program." Our translation deviates from this since in practice all nontrivial quantified expressions would not be well-defined.[3] This means that a JML expression `(\forall Object o;` B`)` is translated to $\forall\ \mathit{Object}\ o; (o.\mathit{created} \doteq \mathit{TRUE} \wedge o \not\doteq \mathtt{null} \rightarrow \lfloor B \rfloor)$.

8.1.2.2 Integer Expressions

Operations on integers in JML are the same as in Java with two exceptions: Expressions with side effects such as `x++` are not allowed in JML. The only addition is that expressions can be of type `\bigint`, on which arithmetic operators represent their mathematical counterparts. Depending on the promoted result type of the compound expression, there are up to three different translations: one each for types `int`, `long`, and `\bigint`. The promoted type is the least restrictive type of the subexpressions (see [Gosling et al., 2013, Sect. 5.1.2]). Other integral types do not occur; there is always an implicit promotion to `int`.

Arithmetic expressions of type `\bigint` are translated to their mathematical counterpart. E.g., n `+` m is translated to $\lfloor n \rfloor + \lfloor m \rfloor$ if at least one of n or m is of type `\bigint`. For the Java types `int` and `long`, the translation relies on dedicated functions which represent the respective modulo semantics; n `+` m is translated to either $\mathit{javaAddInt}(\lfloor n \rfloor, \lfloor m \rfloor)$ or $\mathit{javaAddLong}(\lfloor n \rfloor, \lfloor m \rfloor)$. As with the type restriction predicates above, these proxy functions have different semantics depending on the options in use. The division and modulo operators applied to `\bigint` are translated to the functions *jdiv* and *jmod*,[4] respectively.

Bitwise operations are also translated for `int` and `long` (i.e., operations on the 32- and 64-bit vector types), but it is a type error to use them with `\bigint`. The full table of translations can be found in Table A.9 in the appendix. It is possible to use explicit conversions to enforce certain semantics; e.g., `-`n where n is of type `int` is translated to $\mathit{javaUnaryMinusInt}(\lfloor n \rfloor)$ while `-(\bigint)`n is translated to $-\lfloor n \rfloor$.

[3] In JML, there is no way of expressing createdness of objects. At some point an explicit `\created` operator had been proposed and it was used in older versions of the KeY system, but it has never found its way into the reference manual.

[4] The functions *jdiv* and *jmod* represent division and remainder according to Java rules, albeit in the unbounded domain. They must not be confused with the / and % operators in JavaDL, which represent Euclidian division and modulo; see Section 5.4.

8.1.2.3 Generalized Quantifiers

JML features so-called generalized quantifiers,[5] which include sum and product comprehensions as well as minimum and maximum operators. Syntactically, all of them bind a (logic) variable of some type and consist of an optional Boolean guard expression and a body expression of type **`int`**, both of which the bound variable may appear in. Sum and product comprehensions are not always total functions—consider, e.g., $\sum_{i \in \mathbb{N}} i$. For this reason, JavaDL provides dedicated *bounded* comprehension operators over integer intervals, for which induction schemata can be given (see Section 5.4.2). For bounded comprehensions to be translated, we restrict them to conform to a shape like `(\sum T x; n <= x && x < m; t)` with only one bound variable and an interval which is closed to the left and open to the right, and where T is an integral type. Of course, this excludes certain (well-defined) comprehensions, because of their syntactical shape, such as `(\product Object o; false; 42)` or `(\sum \bigint i, j; 0 < i && i < j && j < 23; i*j)`. Such comprehensions are translated to the unbounded sum and product operators sum and prod, for which only minimal reasoning support is available.

Bounded sum and product comprehensions in JavaDL represent iterated addition or multiplication in the mathematical integers. In JML, the type of a generalized quantifier is the type its body. For a faithful translation, an additional cast is applied to the bounded comprehensions. The expression `(\sum int x; n <= x && x < m; t)`, for example, is translated to the following:

$$castToInt(bsum\{int\ x\}(\lfloor n \rfloor, \lfloor m \rfloor, \lfloor t \rfloor))$$

Minimum and maximum operators appear in the form `(\max T i; A; t)`, which intuitively stands for ‘the maximum of all $t(i)$ such that $A(i)$ holds.’ However, maximum is not a total function either. Consider, e.g., `(\max \bigint i; i)`, for which the above axiomatization would entail that there exists a largest integer. Minimum and maximum operators are translated to dedicated operators in JavaDL, for which there exists only minimal reasoning support at the time of writing;[6] using the **`\min`** and **`\max`** is discouraged. Minimum and maximum can instead be formalized in first-order logic with basic arithmetic. Since it allows for complete reasoning, this is the preferred way in practice.

8.1.2.4 Pure Method Calls

Methods declared as **`pure`** can be used as specification expressions. In JavaDL, pure methods are represented by observer symbols (see Definition 9.7). An instance method call `o.m(p1, ..., pn)` is thus translated to $C{::}m(\mathtt{heap}, \lfloor o \rfloor, \lfloor p_1 \rfloor, \ldots, \lfloor p_n \rfloor)$

[5] See also [Mostowski, 1957] on the concept of *generalized quantifiers* in logic.

[6] In particular, the property described in the JML reference manual [Leavens et al., 2013, Sect. 12.4.24.2] is not provable, where the maximum over an empty range is defined as the minimum over the body type (which is also undefined for `\bigint`).

where C is the class of which $\lfloor o \rfloor$ is an instance containing the most specific method implementation for `m` according to the dynamic dispatch rules of the Java language. For static methods, the receiver parameter is `null`. Not all methods may be used in specifications: Since evaluating the specification must not change the execution context, only *pure* methods may be referred to from JML clauses. Note that in JML, methods that occur in specifications may also be *weakly pure*, i.e., they may create new objects on the heap and change their state, but do not have an influence on the existing part of the heap. The translation to JavaDL thus ignores possible side effects.

8.1.2.5 Referring to the Prestate

In postconditions of method contracts and in history constraints the expression `\old(x)` is used to denote the prestate value of `x`. There is no restriction on the type or syntactical structure of `x` in general; it may include pure method calls or object references. In JavaDL, this can be achieved by performing every heap access which appears in the scope of `\old` with the prestate heap $\mathtt{heap}^{pre}$ instead of the default `heap`. This applies to both field accesses and observer symbols such as pure methods or model fields. E.g., the reference expression `\old(o.f.g)` is translated to $select_T(\mathtt{heap}^{pre}, select_{T'}(\mathtt{heap}^{pre}, o, \mathtt{f}), \mathtt{g})$ and `\old(o.f).g` is translated to $select_T(\mathtt{heap}, select_{T'}(\mathtt{heap}^{pre}, o, \mathtt{f}), \mathtt{g})$. A pure method call as in `\old(this.m())` is translated to $C::m(\mathtt{heap}^{pre}, \mathtt{self})$.

This implementation is an improvement over older versions of KeY which did not use an explicit heap, but replaced occurrences of `\old` with fresh variables which were assigned prior to symbolic execution [Baar et al., 2001]. Neither was it allowed to access pure methods in the prestate.

Note that `\old` can only be applied to proper expressions. This means that JavaDL terms like $select_T(\mathtt{heap}^{pre}, select_{T'}(\mathtt{heap}, o, \mathtt{f}), \mathtt{g})$ cannot be expressed in JML—at least not without jumping through hoops like adding model methods. The obvious `o.f.\old(g)` is not a well-formed JML expression because the reference suffix `g` is not an expression. The generalized version of `\old` with a label to refer to an arbitrary heap state (not just the prestate) is currently not supported in KeY.

Two Notions of the Past

Yi et al. [2013] propose another notion of referring to the prestate. While `\old` stands for a *value* (which may be of a reference type), the proposed `\past` operator represents a *pointer* into the prestate heap. This means that every expression using this pointer is implicitly evaluated in the prestate, e.g., `\past(o).f.g` or `\past(o.f).g` both mean the same as `\old(o.f.g)`. The main motivation for such an operator is to bridge a gap with `\old` which exposes implementation detail. Imagine `o`'s static type to be an interface. How do we state that the object denoted by `o` in the poststate equals `o` in the prestate without exposing implementation details? Using `\past`, this can be expressed as `o.equals(\past(o))`, but there does not exist an equivalent expression using `\old`. Please note that the *value* of `\past(o)` is still the same as `\old(o)`.

Even on the level of JavaDL, this is difficult to express. While not in standard JML, KeY's extension features two-state model methods (see Section 9.2.2). These represent observer functions which observe two heaps simultaneously. This allows the `\old` operator to appear in the implementation. We could give a two-state model method `equalsOld()` with the following implementation:

```
/*@ public two_state model boolean equalsOld ()
  @     { return this.f == \old(this.f); }
  @*/
```

However, we would have to implement such a model method for each concrete subtype because the implementation refers to the fields of the concrete type. Note that model methods are always *strictly* pure in our JML dialect.

The Boolean expression `\fresh(o)` also appears in postconditions and states that `o` points to a freshly allocated object, i.e., it was not created in the prestate and it is not a null reference:

$$\lfloor \texttt{\textbackslash fresh}(o) \rfloor = select_{boolean}(\texttt{heap}^{pre}, \lfloor o \rfloor, created) \doteq FALSE \land \lfloor o \rfloor \not\doteq \texttt{null}$$

Note that the value of o is evaluated in the poststate. Please note that the `\fresh` operator is overloaded; there is an expression `\fresh(s)` where `s` is a location set expression in KeY's dialect of JML, which means that all locations in `s` belong to objects which were newly allocated.

8.1.2.6 Type Expressions

Standard JML features a type of types `\TYPE`, which is not present in KeY's JML since the underlying JavaDL assumes a finite type system. Type expressions as such are supported within certain contexts: Boolean expressions of the form `\typeof(x) == \type(T)` where `x` is an expression of any type and `T` is a type, are translated to $exactInstance_T(x)$ (introduced in Section 2.4.3). Only the syntax where a `\typeof` expression appears on the left hand side and `\type(T)` (denoting a fixed type) appears on the right hand side is supported. Any other occurrences of type equality are Skolemized.

To describe that an expression x evaluates to an instance of type T, but not necessarily to an exact instance of T, the Java operator `instanceof` can be used. The expression `x instanceof T` is translated to $instance_T(\lfloor x \rfloor)$.

8.1.2.7 Location Set Expressions

Weiß [2011] introduced dedicated location set expressions to JML. For some of them a translation is straightforward, as they have been designed to correspond to predicates and functions in JavaDL with obvious meaning, e.g., `\intersect(s,t)`. But location set expressions also replace *reference set expressions* from standard JML. These are faithfully translated to terms in JavaDL. For instance, $\lfloor$`\everything`$\rfloor = \mathit{setMinus}(\mathit{allLocs}, \mathit{unusedLocs}(\mathtt{heap}))$, taking into account that JML only considers locations which belong to already *allocated* objects. Please note that the keyword `\strictly_nothing` (an extension introduced by KeY) is not an expression in this sense, but can be used to form a nonstandard `assignable` clause, see Definition 8.4 below.

The binary union operator is called `\set_union` for technical reasons. The JML language also features a set comprehension operator `\infinite_union` that binds a variable of any type and has a location set expression in the body. Optionally, a guard can be given. Like other comprehension operators, the translation from JML to JavaDL includes default guards. For instance, the JML expression `\infinite_union(Object o; \singleton(o,f))` is translated to the following term:

$$\begin{aligned}\mathit{infiniteUnion}\{\mathit{Object}\ o\}(&\text{if } (x.\mathit{created} \doteq \mathit{TRUE} \land x \neq \mathtt{null})\\ &\text{then } (\{(o,\mathtt{f})\}) \text{ else } (\mathit{empty}))\end{aligned}$$

The set comprehension notation of standard JML is not supported in KeY.

8.1.2.8 Reachability

Both standard JML and the dialect used by KeY feature a `\reach` operator, but their syntax and semantics differ. Both serve the purpose of specifying properties on the set of objects (excluding `null`) which are reachable by subsequent field and array index references. In standard JML, `\reach(o)` intuitively stands for the set[7] of all objects transitively reachable through any instance field from the reference `o`.

By contrast, in KeY `\reach` is a predicate symbol that states whether an object is reachable from another one. It takes as a parameter the locations that are allowed in the reference chain—including static fields. The operator appears both as 3-place and 4-place, where `\reach(`ℓ`, `o_1`, `o_2`)` means '$\lfloor o_2 \rfloor$ is transitively reachable from $\lfloor o_1 \rfloor$ through any location in $\lfloor \ell \rfloor$,' where ℓ is a location set expression; and `\reach(`ℓ`, `o_1`, `o_2`, `n`)` stands for reachability in *exactly* $\lfloor n \rfloor$ steps. The former is equivalent to `(\exists \bigint n; n >= 0; \reach(`ℓ`, `o_1`, `o_2`, n))`.

Except for the fact that there is no explicit reasoning about sets of objects in KeY, its reachability operators are more expressive than those of standard JML. The standard JML expression `\reach(`o_1`).contains(`o_2`)` is equivalent to

[7] More precisely, it is an object of type `JMLObjectSet` since JML does have abstract data types.

`\reach(`o_1`.*, `o_1`, `o_2`)`, while nontrivial location sets cannot be expressed in standard JML.

A similar operator is `\reachLocs` that denotes a location set consisting of all locations of reachable objects. Again, there are two versions of `\reachLocs`, one with an explicit number of steps and one with implicit quantification.

8.1.2.9 Escaping to JavaDL

It may happen that some properties cannot (or at least not without considerable effort) be represented in JML, but can be represented on the level of JavaDL. A typical case are user-defined functions or predicates which do not have a counterpart on the JML level.[8] For this purpose, KeY introduces escapes from JML into JavaDL. Within the delimiters `(* *)`+ (known as "informal predicate" in standard JML) any JavaDL term may appear, which is inserted verbatim during translation. Even more convenient is the function escape `\dl_`, which allows one to refer to a nonbinding JavaDL function (or predicate) while parameters are still given in JML. The escape sequence `\dl_` must be immediately followed by a function name. Variable binding is not allowed.

For instance, `\dl_add(a,\old(a))` refers to the function *add*, which represents addition in the mathematical integers. This function is not directly available in JML when the parameters have Java integer type. In case the JavaDL function has a heap parameter the base heap `heap` is implicitly added as the first parameter. Take a function $f : \mathit{Heap} \times \mathit{Object} \to \mathit{Object}$, for instance. `\dl_f(`$o$`)` is translated into $f(\mathtt{heap}, \lfloor o \rfloor)$. JML operators such as `\old`, whose translation to JavaDL can be tedious to express,may be used in parameters.

8.1.3 Abstract Data Types in JML

KeY's extension to JML additionally features the *abstract data type* [Reynolds, 1994] of finite sequences at the language level, referred to as `\seq`. This type is *primitive* in Java lingo, like the other specification-only types `\bigint` and `\locset` (see above). Reasoning about the underlying theory of finite sequences is well supported in KeY (see Section 5.2).

Algebraic data types can be defined *inductively*, i.e., their definition consists of a definition for each constructor. This kind of recursive definition is both well-founded and total due to the inductive nature of initial algebras [Jacobs and Rutten, 1997] that entails that every element of the carrier set can be uniquely described using a finite number of constructor applications (i.e., construction is invertible). As an example, in the `List` example above, we can model each state of the list using only the two

[8] Model methods (see 9.2.2) may instead be used for specification. However, in *reasoning*, model methods are treated similarly as (pure) Java methods, while functions or predicates can be given dedicated rules to reason about them efficiently.

constructors 'empty list' and 'appending an element,' which form a basic sequence data type. This principle also allows us to do proofs by induction. The length can be defined as an observer of these constructors. We can then perform induction over the length of a sequence.

The algebraic data type `\seq` of finite sequences is predefined in KeY-JML, its operations are displayed in Table 8.1. These operators are directly translated to their counterparts in JavaDL. Section 5.2 presents the underlying theory of finite sequences. In particular, we have a comprehension operator `\seq_def` where (`\seq_def \bigint x;` i; j; t) denotes the sequence $\langle t[x/i],\ldots,t[x/j-1]\rangle$. Please note that `\seq` is not a parametric type; its elements are not typed. For this reason, sequence access always needs to be preceded by an (unsafe) type cast.[9]

Table 8.1 Defined operations on the `\seq` data type in JML (extension in KeY)

	syntax	signature
empty sequence	`\seq_empty`	$\to$`\seq`
singleton sequence	`\seq_singleton(`e`)`	$T \to$ `\seq`
concatenation	`\seq_concat(`$s1$`,` $s2$`)`	`\seq`$\times$`\seq`$\to$`\seq`
subsequence	s`[`i`..`j`]`	`\seq`$\times$`\bigint`$\times$`\bigint`$\to$`\seq`
comprehension	`(\seq_def \bigint x;` i`;` j`;` t`)`	`\bigint`$\times$`\bigint`$\times T \to$`\seq`
access	`(`T`)`s`[`i`]`	`\seq`$\times$`\bigint`$\to T$
length	s`.length`	`\seq`$\to$`\bigint`

Like `\bigint` or `\locset`, the type `\seq` counts as a primitive type in the Java sense. This means that all operations are side effect-free like mathematical functions, instances do not need to be created, and expressions can be compared using equality (`==`). In particular, it is allowed to quantify over all (infinitely many) sequences. Abstract data types must be distinguished from *model types* [Leavens et al., 2006b, Sect. 2.3] in standard JML, which are not supported by KeY. These model types—like `JMLObjectSequence`—still are Java reference types that may be used in specifications—with all their issues like createdness.

8.1.4 Well-Definedness of Expressions

Some functions or predicates are only partially defined. A standard example is the division function which is only defined for divisors other than zero. In the context of Java programs, illegal heap accesses are particularly important, e.g., the value of a field access on `null` is not defined, as is the value of `a[5]` where `a` is an array of length 5 or less. According to the JML reference manual [Leavens et al., 2013], a Boolean expression is valid in a state if it has the truth value true and "does not cause an exception to be raised."

[9] In JavaDL, the access function itself is type parametric. An access in JML (prefixed with a type cast) is translated to the appropriate typed access. See Table A.15 in Appendix A for details.

Our translation from JML to JavaDL ignores this dimension of undefinedness. KeY can generate well-definedness proof obligations (see Section 8.3.3 below) that establish well-definedness of JavaDL formulas as described by Kirsten [2013].

8.2 From JML Contract Annotations to JavaDL Contracts

In this section we introduce JavaDL contracts as the principal concept in the verification framework of KeY. First and foremost, JavaDL contracts serve as an intermediate layer between JML specifications and proof obligations in JavaDL. The largest part of this section is taken by defining a normalization of JML contracts comprising various steps (Section 8.2.1). Then the special cases of contracts for constructors (Section 8.2.4) and model methods and fields (Section 8.2.3) are covered and finally the formal definition of a JavaDL contract—and how it is derived from a JML contract—is given in Section 8.2.4. The subsequent Section 8.3 then describes how the JavaDL proof obligations for the correctness of a JML specification are constructed. It will be explained in Chapter 9 how contracts can be used in proofs for sound modular reasoning about Java programs.

The definition of a proof obligation encompasses more than a mere translation of the JML expressions in the clauses of the contract into JavaDL. Additional logical constructions are needed to model aspects of the Java world precisely in the first-order setting of JavaDL. We add constraints to confine the liberal model of general predicate logic to those system states which can be reached through the execution of Java code. For instance, in Java a field with a reference type can only point to either `null` or to an already created object, but in JavaDL, it could possibly also point to an object yet to be created. The proof obligations for methods use dynamic logic constructs of JavaDL as they need to talk about both the before- and the after-state of execution of methods.

We discuss in general how the contracts for a generic method are handled in KeY. For this sake we assume that a method `m` is defined in some class *C* as follows.

```
class C {
    public R m (final T1 p1, ..., final Tn pn) { ... }
    ⋮
}
```

We assume that all parameters $\mathtt{p}_i$ are declared `final`, i.e., they are not assigned a value in the method body.[10]

[10] This restriction is not present in the KeY system, but it eases the presentation.

8.2.1 Normalizing JML Contracts

JML is a feature-rich specification language in which the same specification intention can often be formulated in different ways. This eases the job for the specifier and makes specifications more concise and easier to understand.

For instance, JML allows the formulation of structured specifications. The behavior of a method does not need to be formulated as a single contract, but can be split up into multiple, possibly nested individual contracts (called *specification cases*) that model different parts of the behavior. Within a contract, multiple clauses of the same kind (e.g., several `ensures` clauses) can be used to express properties of the behavior; keywords like `normal_behavior` or `pure` can be used as abbreviations of frequently applied specification elements. Moreover, JML is designed as a redundant language in which many features have more than one associated keyword.

The syntactic richness of the specification language is a benefit when readability and understandability of specifications is desired. However, for the precise description of the translation of contracts, a small core language having the same expressiveness, is favorable. In the following, we consider such a core language[11] for JML in which additional specifications constructs are assumed *syntactic sugar* defined in terms of that core. The considered JML core language closely resembles the one presented by Raghavan and Leavens [2000], although we deliberately deviate in some respects.

We present a normalization process that translates a general JML method contract without syntactical restriction into a *normalized JML contract* in the core language. This 'desugaring' may yield one or more separate contracts[12] for the given method, of which it needs to satisfy all. Note that this transformation is only used as a concept for explanation; JML contracts are not implemented in this way in the KeY system.

The JML normalization process consists of the following steps:

1. Flattening of nested specifications
2. Making implicit specifications explicit
3. Processing of modifiers
4. Adding of default clauses, if not present
5. Contraction of multiple clauses
6. Separation of verification aspects

We consider two classes of normalized contracts: functional contracts and dependency contracts. Listing 8.1 displays the shape of a normalized functional method contract as we produce it in this section, while Listing 8.2 displays the shape of a normalized dependency contract. For details on JML clauses, see Section 7.1.1.

In the next paragraphs we outline the ideas behind the normalization steps. They may be skipped by readers familiar with the semantics of the desugared JML con-

[11] Our idea of a 'core' is to include a minimal syntax that has enough expressive means to accommodate the meaning of the entire language as we support it.

[12] We say here that a method can have more than one contract since that fits best the translation into JavaDL. Within the JML community it is more common to say that every method has *precisely one* contract with possibly several cases (including those inherited from supertypes). The difference is only terminological, not conceptual.

```
/*@ M behavior
  @ requires Pre;
  @ ensures Post;
  @ signals (Throwable e) ExPost;
  @ diverges [true|false];
  @ measured_by Var;
  @ assignable Ass;
  @ helper
  @*/
/*@nullable*/ RetType methodName(/*@nullable*/ T1 p1, ... )
```

Listing 8.1 JML functional method contract specification case template

```
/*@ M behavior
  @ requires Pre;
  @ measured_by Var;
  @ accessible Acc;
  @ helper
  @*/
/*@nullable*/ RetType methodName(/*@nullable*/ T1 p1, ... )
```

Listing 8.2 JML method dependency contract specification case template

structs and who are convinced that the shape of the normalized contract is general enough.

8.2.1.1 Flattening of Nested Specifications

JML allows the specification of nested cases (also called structured specifications) using the `{| ... |}` construct with opening and closing braces. It can be used to formulate specifications with some common clauses which are relevant for all cases, and with clauses for several separate and specific cases. The listing on the left of

```
                                        before
                                        alt1
                                      also
                                        before
                                        alt2
   before                             ...
   {|     alt1                        also
     also alt2   ...                    before
     also altn  |}                      altn
```

Figure 8.1 A nested JML specification (on the left) and the flattened contracts (on the right) after expansion

Figure 8.1 depicts the syntactical form of a nested specification where *before* is a (possibly empty) sequence of **`requires`** clauses and alt_i is a sequence of arbitrary JML clauses (possibly including further **`requires`** clauses). The intuitive meaning of the nested clauses is that *any one* of the clauses connected by **`also`** makes a valid contract (but not the 'outside' preconditions on their own). The nonnested specification cases can thus be derived by replacing `{| ... |}` by any one of the alternatives alt_i. The nested contract in the listing on the left of Figure 8.1 is, hence, equivalent to the list of the n separate specification cases (conjoined using the keyword `also`) that appears in the listing on the right. This expansion can be performed in the same manner when more than one nesting operator occurs, or if the nesting of cases is nested itself.

For the remaining desugaring steps, we consider the separated flat contracts individually.

8.2.1.2 Making Implicit Specifications Explicit

JML provides a number of modifiers and specific keywords for frequent specification scenarios. For the description of the translation, however, it is advisable to make their meaning explicit by means of other specification clauses to reduce the number of cases that need to be considered. Section 7.5 describes how the JML user can specify whether a method parameter or its return value may take the value **`null`**.

Making Nonnull Specifications Explicit

JML follows a 'nonnull by default' policy (see also Sections 7.5 and 8.1.1) which means that every reference type in a method declaration (type of a parameter or return type) which is not explicitly annotated with the JML modifier `/*@`**`nullable`**`*/` is implicitly declared as nonnull. In a first normalization step, we make these implicit assumptions explicit by adding `/*@`**`non_null`**`*/` in those places without explicit nullity annotation.[13]

Then we make the semantics of the nullity modifiers explicit by replacing every `/*@`**`non_null`**`*/` modifier in front of a method parameter `p` by `/*@`**`nullable`**`*/` and at the same time add the clause **`requires`** `p != ` **`null`**`;` to every method contract for the method. If the return type of a method is `/*@`**`non_null`**`*/`, we also replace that modifier by `/*@`**`nullable`**`*/` and add the clause **`ensures`** `\result != ` **`null`**`;` to every contract for the method. These steps do not change the semantics of the contracts, but make it explicit.

In case the type of a method parameter (or the return type) is an array type over a reference type (e.g. `Object[]`), the nonnull annotation does not only specify that the value is always different from **`null`**, but also that all entries differ from **`null`**, too. For arrays of higher dimension this goes even deeper. To specify this, we introduce the JavaDL predicate $nonNull : Heap \times Object \times int$. The formula $nonNull(h,x,d)$ is

[13] Unless the enclosing class has been annotated with `/*@nullable_by_default*/`, in which case `/*@nullable*/` is the added modifier.

true if and only if x refers (on heap h) to an array of objects different from `null` that themselves are nonnull arrays of dimension $d-1$. Formally, it is defined through the following axiom.

$$\begin{aligned}&\forall \mathit{Heap}\ h, \mathit{Object}\ x;\ \mathit{nonNull}(h,x,0) \leftrightarrow x \not\doteq \mathtt{null} \land\\ &\forall \mathit{Heap}\ h, \mathit{Object}\ x, \mathit{int}\ d;\ d>0 \rightarrow (\mathit{nonNull}(h,x,d) \leftrightarrow x \not\doteq \mathtt{null} \land\\ &\qquad (\forall \mathit{int}\ i;\ 0 \le i \land i < x.\mathit{length} \rightarrow \mathit{nonNull}(h, \mathit{select}_{\mathit{Object}}(h,x,\mathit{arr}(i)), d-1))\ .\end{aligned}$$

For a d-dimensional array parameter x declared as `/*@non_null*/ Object[]`^{d} `x`, the precondition then reads `requires \dl_nonNull(x, `d`);`.[14]

Making Object Invariant Specifications Explicit

Like the nonnullness of method parameters, receiver class invariants are also part of the specification without being explicitly written down.

In standard JML the objects for which the class invariants hold are determined by the so-called *visible state semantics*; in KeY's JML, all objects for which the class invariants hold must be stated explicitly using the operator `\invariant_for`. With one exemption: A nonstatic method is implicitly assuming (as a precondition) the invariant for the receiver object `this` before the method call and needs to assure it after the call (as a postcondition). This default specification can be explicitly deactivated by adding the modifier `/*@helper*/` to the method specification.

To desugar the implicit invariant semantics for nonhelper methods, we add a `helper` modifier to the method and the clauses

- `requires \invariant_for(this);`
- `ensures  \invariant_for(this);` and
- `signals  (Throwable e) \invariant_for(this);`

to every specification case for the method. A static method has no receiver object, and thus cannot refer to an object invariant. Instead, the *static* class invariant is implicitly assumed and must be guaranteed. The translation as explicit clauses[15] reads `requires \static_invariant_for(`C`);` in which C is the enclosing class in which the static method defined.

The order of clauses plays an important role in the well-definedness of contracts (see Section 8.3.3). It is therefore important to mention that the newly added clauses are added *before* the first existing annotation.

[14] The first argument to `\dl_nonNull` of type *Heap* is added automatically by the translation as outlined in Section 8.1.2.9.

[15] The operator `\static_invariant_for(`C`)` referring to the static invariant of class C is a KeY extension to JML.

Making The Kind Of Behavior Explicit

JML supports specification not only of normally terminating program runs, but also for the case of abnormal termination (uncaught exceptions). When writing a specification, one can distinguish between specification of the normal and of the exceptional case by declaring them as **`normal_behavior`** and **`exceptional_behavior`**, respectively.

For a normalized contract, both keywords are reduced to the keyword **`behavior`** by which a contract is initiated. The normal behavior gets an additional clause `signals (Throwable t) false;` indicating that the method does not raise any exception or error. Likewise, exceptional behavior specifications get an additional postcondition `ensures false;` indicating that the method never terminates normally. Note that the declaration of either behavior does not specify divergence.

Desugaring `signals_only` Clauses

KeY supports **`signals_only`** clauses, which restrict the types of exceptions that can possibly be raised by a method. Unlike the **`throws`** declaration in the Java language, it does not only constrain *checked exception* types (subclasses of `Exception` which are not subclasses of `RuntimeException`), but all instances of class `Throwable`. For a discussion on exception types in Java, see the box on page 260.

The clause **`signals_only`** T_1, …, T_p; lists one or more names T_i of classes extending `Throwable`. It can be replaced by the semantically equivalent clause:

`signals` `(Throwable e) (e` **`instanceof`** T_1 `||...|| e` **`instanceof`** T_p`);`

8.2.1.3 Expanding Purity Modifiers

There are two more method modifiers **`pure`** and **`strictly_pure`** indicating that a method does not have (observable) side effects. They both mean that the method terminates unconditionally and that it does not modify existing heap locations. The modifier **`pure`** is hence translated into the two clauses `diverges false;` and `assignable \nothing;`. The modifier **`strictly_pure`** is an extension introduced to JML by KeY to indicate that the heap is not modified at all (neither existing nor freshly created locations; see also Section 7.9.1). It becomes `diverges false;` and `assignable \strictly_nothing;` when translated into JavaDL. The semantic differences between **`pure`**/`nothing` and **`strictly_pure`**/`strictly_nothing` are outlined in Section 8.2.4.

8.2.1.4 Adding Default Clauses

The clauses in the normalized contract in Listing 8.1 are not optional. If a contract does not have (at least) one clause for every keyword, clauses with default values

Table 8.2 Default values for absent clauses and operators used to contract two or more clauses with the same keyword

Clause	Default value	Contraction operator
`requires`	`true`	`&&`
`ensures`	`true`	`&&`
`diverges`	`false`	`||`
`assignable`	`\everything`	`\set_union`
`accessible`	`\everything`	`\set_union`
`signals`	`(Throwable t) true`	*see below*
`signals_only`	*see below*	*not allowed*
`measured_by`	*not specified*	*not allowed*

are added to make the contract complete. The second column in Table 8.2 lists the default values which are used for the clauses added in case a keyword does not occur.

Default values are designed in such a fashion that they match the user's expectations of an unconstrained method, known as the *principle of least surprise* [Leavens, 1988]. The default clauses express that the method may be called in any state and that it may terminate in any state. It may also terminate abnormally with any exception or error; it may read from or write to any location on the heap.

The default clause for **diverges** is a little different in this context since its default is to disallow nonterminating behavior. Instead, if nontermination is to be allowed for a specific method (e.g., for the event loop of a reactive system), it must be explicitly stated. In this respect, the default value is not the most liberal, but rather the most restrictive one. It matches user expectation, however, since more often than not *do* we want our code to terminate.

As described above, clauses of type **signals_only** are desugared. We define a default value in case no such clause is given, even though it will be translated into a **signals** clause. The default for **signals_only** clauses includes the *unchecked* exception types `Error` and `RuntimeException` as well as those *checked* exception types that are explicitly declared in the **throws** clause of the method signature. This is the most liberal specification possible in Java since these are all the exception types that the compiler permits to be thrown. For a method with the signature `void foo() throws IOException`, for instance, the default clause is

```
signals_only Error, RuntimeException, IOException; .
```

Exceptions and Errors

In Java methods one may throw exceptions and errors to indicate abnormal situations and to terminate execution abruptly. Java discriminates between *regular exceptions* (i.e., instances of `java.lang.Exception`) and *errors* (i.e., instances of `java.lang.Throwable` that are not instances of `Exception`). While the former are designed to be handled within the program (to recover from the abnormal

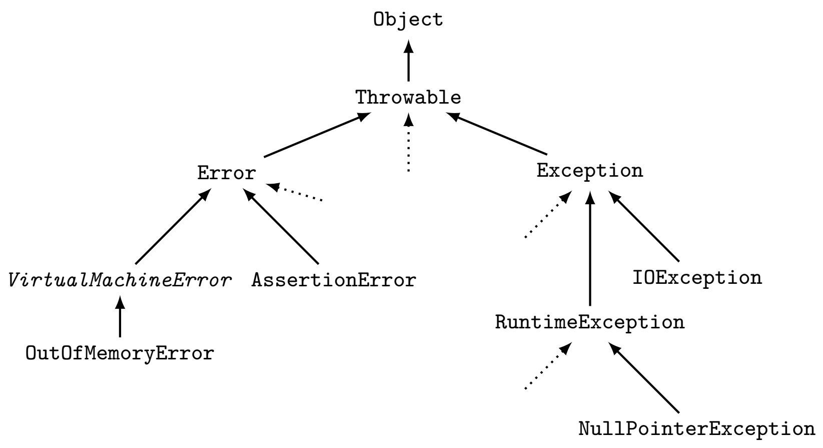

Figure 8.2 The type hierarchy of exceptions in Java

situation), the latter are reserved for severe, unexpected internal problems. Errors are not meant to be caught but to terminate the whole program abruptly. A typical example for an error is `OutOfMemoryError` that is thrown by the virtual machine if a memory allocation fails due to lack of (physical) memory.

Both, regular exceptions and errors, have *unchecked* exceptions as subtypes, that may be raised at any time during execution without the been to declare them at compile time. Unchecked regular exceptions are instances of `java.lang.RuntimeException`, unchecked errors are instances of `java.lang.Error`. All other exception types are *checked* exceptions. An excerpt from the Java type hierarchy is shown in Figure 8.2.

In the JML view of things, an execution which terminates abnormally by a thrown exception is still within the scope of the specification. JML distinguishes between normal postconditions (specified using **`ensures`**) and exceptional postconditions (specified using **`signals`**).

The situation is different for errors: The JML reference manual [Leavens et al., 2013, Sect. 9.6.2] defines any method contract to be fulfilled vacuously if the method terminates with an error. On the one hand, errors may appear at many occasions during execution and in an unpredictable (and in some sense nondeterministic) manner; hence, it may be justified to ignore them. On the other hand, an error represents a severe failure of the software system that must not be overlooked (see [Bloch, 2008, Item 57f.]).

One pathological example that shows that ignoring errors is problematic is the following method which employs the Java **`assert`** statement (not the JML equivalent in JML comments). If the asserted property is not met, an assertion in Java code raises an `AssertionError`. Hence, the following JML method contract is valid according to the JML reference manual [Leavens et al., 2013]:

```
    //@ normal_behavior
    //@ ensures false;
    void foo () { assert false; }
```

This is surprising since the contract is intuitively unsatisfiable. It can be argued that the semantics in the JML reference manual [Leavens et al., 2013]is mostly motivated by runtime assertion checking, not by static verification, and therefore does not need to be concerned with errors. Since, however, the Java language actually allows the programmer to raise arbitrary instances of `Throwable` (and its subclass `Error`)—and also to catch them—it is reasonable to extend the semantics of exceptional contracts to embrace errors as well as regular exceptions.

In KeY, JML **`signals`** clauses may list any subclass of `Throwable`. This is vital for the soundness of the contract framework of KeY (see Section 9.1.3). Otherwise, a caller of the above method could rely on the (unsatisfiable) postcondition after catching the error.

The actual causes of unpredictable errors (insufficient main memory, too deeply nested recursions, incompatible class files, etc.) could be modeled in a static analysis and be reasoned about. This would increase the verification cost tremendously, however. In KeY, all such error causes are hence silently ignored.

The **`measured_by`** clauses do not have a default value. If it is not specified, that aspect of the specification is left open. Unless the method calls itself recursively (directly or indirectly via intermediate method calls), this clause is not required.

8.2.1.5 Contraction of Clauses

The normalized contracts of Listings 8.1, 8.2 not only require at least one clause for every keyword, but also that there be *at most* one. Prior to normalization, there may be several clauses of the same kind in a contract, which helps structuring the specification. It is, for instance, considered good practice to specify each aspect of the precondition in its own **`requires`** clause.

In cases where multiple clauses of the same kind have been specified, they must be contracted to one single clause. The operator used to contract two or more clauses into one clause depends on the kind of the clause. The operators used for the respective clause types are listed in the right column of Table 8.2. Pre- and postconditions, for instance, are both connected using the logical conjunction `&&`. Note that due to lazy evaluation in Java, the order of clauses matters for well-definedness as we will point out in Section 8.3.3.

The contraction of **`signals`** clauses is a little more delicate since they may give postconditions for various exceptional situations. The two clauses

```
signals (ExcClass1 e) Post1;
signals (ExcClass2 e) Post2;
```

can be contracted to the semantically equivalent single **`signals`** clause

```
signals (Throwable e)  (e instanceof ExcClass1 ==> Post1)
                    && (e instanceof ExcClass2 ==> Post2) .
```

Depending on the type of the exception by which the method is terminated, the respective postcondition *must* hold.

For `signals_only` and `measured_by` clauses, multiple specifications do not make sense and are, hence, not allowed.

8.2.1.6 Separation of Verification Aspects

The contract language of JML is rich and the specifications may cover several behavioral aspects at the same time. We now describe how a single contract touching on more than one specification aspect is broken down into different single-aspect contracts.

Separation Of Functional And Dependency Contracts

A contract at this point may still have both functional clauses (that appear in Listing 8.1) and dependency clauses (that appear in Listing 8.2).

These are separated into the two categories: The functional clauses (`signals`, `diverges`, `ensures` and `assignable`) constitute the functional contract whereas the `accessible` clause makes up the dependency contract. The `requires` and `measured_by` clauses are shared by both.

If one of the functional or the dependency contract is trivially fulfilled (for instance if `accessible \everything` is specified), that trivial contract is dropped immediately.

Splitting Possibly Diverging Contracts

JavaDL can only handle either partial or total contracts and does not have a concept of contracts for conditional termination. Therefore, any contract with conditional termination is transformed into two unconditional contracts such that a contract that contains `diverges` d; becomes the two cases

```
requires d;         requires !d;
diverges true;      diverges false;
```

unless it is a constant (`true` or `false`) already. We will assume this shape of contract from now on.

8.2.1.7 Example

We illustrate with a small example the result of normalization. Listing 8.3 shows a method specified with a single method contract that contains both functional elements (like the `signals` or the `ensures` clauses) and dependency elements (the `accessible` clause).

```
class Example {
  /*@ public behavior
    @  requires to >= from;
    @  signals_only IndexOutOfBoundsException;
    @  signals (IndexOutOfBoundsException e) from < 0 || to >= a.length;
    @  ensures a[\result] >= a[from];
    @  accessible a[*];
    @*/
  /*@ pure */
  public int maxIntArray(int[] a, int from, int to) {
    // ...
  }
}
```

Listing 8.3 Example of a JML method contract prior to desugaring

Listing 8.4 shows the same method with the two contracts that are the result of the normalization process described above. Semantically, the two specifications are equivalent. It is easy to see that the original specification is much conciser. However, the normalized contracts have no implicit clauses and are easier to handle in logic.

8.2.2 Constructor Contracts

JML contracts can also be annotated to Java class constructors. The normalization process is almost the same as the one described above. But a few differences do exist:

- As the object has only been created just prior to the constructor call, assuming the instance invariant to hold already, is not sensible. Therefore, `\invariant_for(this)` is not an implicit precondition for constructors. It is, however, implicitly added to the (normal and exceptional) postconditions for constructors, because a constructor is obliged to establish initially the invariant of the created object. The static class invariant `\static_invariant_for(C)` is added to any nonhelper constructor of class `C`.
- In contrast to original JML, constructor contracts in KeY's variant of JML are attached to `new` invocations, i.e., the sequence of both instance allocation, field initialization, and the actual constructor execution (see Section 3.6.6).

```
class Example {
  /*@ public behavior
    @  requires a!= null
    @     && \invariant_for(this) && to >= from;
    @  signals (Throwable e)
    @        (e instanceof IndexOutOfBoundsException ==>
    @                  from < 0 || to >= a.length)
    @     && (e instanceof Throwable ==> \invariant_for(this))
    @     && (e instanceof IndexOutOfBoundsException);
    @  ensures a[\result] >= a[from] && \invariant_for(this);
    @  diverges false;
    @  assignable \nothing;
    @ also
    @  requires array != null
    @     && \invariant_for(this) && to >= from;
    @  accessible a[*];
    @*/
  /*@ helper */
  public int maxIntArray(/*@nullable*/int[] a, int from, int to) {
    // ...
  }
}
```

Listing 8.4 Example of the JML method contract from Listing 8.3 after desugaring

- As a consequence, the instance to be initialized is fresh, i.e., `\fresh(this)` is true in the poststate. Likewise, a (weakly) pure constructor may assign the fields of `this`.

8.2.3 Model Methods and Model Fields

JML supports methods which exist for verification purposes only: *model methods*. They reside, like all JML annotations, in special comments and may—as specification artifacts—make use of the language capabilities of JML. The types and expressions used in model methods need not be constrained to those of Java. Model methods can be subjected to contracts in exactly the same way as Java methods. When defining a model method, the modifier `model` must be used to indicate its nature as specification-only element (like for model fields).

A model method to compute the sum of the absolute values of a sequence of integers together with a method contract could thus read:

—— JML ——

```
/*@ public behavior
  @  requires seq.length > 0;
  @  ensures \result >= 0;
  @  assignable \strictly_nothing;
  @ model int sumAbs(\seq seq) {
  @     return (\sum int i;
  @         0<=i && i<seq.length; Math.abs((int)seq[i]));
  @ }
  @*/
```

——— JML ———

Besides model methods, JML also supports the less general, but related, concept of *model fields* (motivated and introduced in Section 7.7.1). Conceptually, model fields can be considered as model methods without arguments. Thus, model fields are far more related to query *methods* than to ordinary Java *fields* since their value is not stored within the heap state space but is *computed* from the heap state. However, on the *syntactic* level model fields are declared like fields and quite differently than model methods. The expressive power of model methods is much higher than that of model fields and will be explained in detail in Section 9.2.2. To reduce the number of syntax elements in normalized annotations, this section reports on how model fields can be reduced to nullary model methods, and proceeds then with model methods.

Model fields have their definition fixed by a **represents** clause. Such clauses are implicitly private, in the sense that the definition given by them applies only to exact instances of the class with the clause; a redefinition (or a repetition of the original definition) is required in subclasses. Represents clauses have an unmodifiable implicit precondition `\invariant_for(this)`; thus, the definition of a model field must only be expanded if the object invariant of the receiver object holds.

The general model field definition

```
/*@ model T modelField;
  @  represents modelField = Repr;
  @  accessible modelField : Acc
  @             \measured_by Var;
  @*/
```

hence is semantically equivalent[16] to the definition of the strictly pure model method

```
/*@ public behavior
  @  accessible Acc;
  @  measured_by Var;
  @  requires \invariant_for(this);
  @  assignable \strictly_nothing;
  @ model T modelMethod() {
  @   return Repr;
  @ }
  @*/
```

in which the represents clause *Repr* has become the value returned by the model method. All references to `modelField` must be replaced by a call `modelMethod()` to the method without arguments. The clause `requires \invariant_for(this);`

[16] See the box on page 267.

has been made explicit[17] to emphasize the fact that the invariant needs to hold when evaluating the model method.

If the value of a model field is not defined using a *functional* predicate **`represents`** but *relationally* using the more general `such_that` mechanism, the model field definition

```
/*@ model T modelField;
  @  represents modelField \such_that Cond;
  @*/
```

becomes as a model method with the semantically equivalent definition

```
/*@ public behavior
  @  requires \invariant_for(this);
  @  ensures Cond[this.modelField → \result];
  @  assignable \strictly_nothing;
  @ model T modelMethod();
  @*/
```

in which *Cond*[`this.modelField` $\to$ `\result`] denotes the JML expression *Cond* in which every reference to **`this`**`.modelField` has been replaced by the keyword **`\result`**.

A model method may—like an abstract method—be declared without specifying a method body. While an abstract method, however, must be refined in a concrete implementation class by a concrete method with a method body definition, a model method may remain underspecified without implementation. It is then the contract of the model method that characterizes the semantics of the symbol, see Section 9.2.2.

Subtle Differences Between Model Methods and Model Fields

Above we claimed that the model method replacement for a model field is semantically equivalent. This is the case when looking at the matter from a distance. Model methods possess some advanced features which make their semantics deviate slightly from model fields:

- *Model method bodies are inherited, but represents clauses of model fields are not.* As for an ordinary method, a model method definition is inherited by all subclasses unless they provide a new method definition. Thus, a subclass not mentioning a redefinition of the model method has the same definition as the superclass whereas the model field remains undefined for instances of the subclass.
- *Termination conditions are different between model methods and fields.* For recursive model fields, a variant must be specified using a **`\measured_by`** statement. Mutually dependent model fields need not provide evidence for termination and formulating inconsistent definitions is thus possible. Model

[17] It would also be assumed implicitly unless `modelMethod` were declared **`helper`**.

methods have a stricter termination model in the sense that there must never occur infinite recursion when evaluating them.

8.2.4 JavaDL Contracts

With normalized JML contracts at hand, it is time to bring the specification language artifacts into the logical context of JavaDL. In the following, we will see how JML contracts are translated into contracts on the level of JavaDL in such a fashion that most of the clauses in a normalized JML contract have a direct counterpart on the JavaDL side. Some of the clauses are contracted on the logical side, when they express aspects of common concern. As with normalized JML contracts, there are separate JavaDL contracts for the functionality of a method (describing the behavioral effects of a method) and contracts for the dependency of a query method (describing which part of the heap a computation may depend upon).

For the reader's convenience we repeat Definition 3.22 of functional method contracts here:

Definition 8.2 (Functional method contract). A functional JavaDL method contract for a method or constructor R `m(`T_1 `p`$_1$`, ..., `T_n `p`$_n$`)` declared in class C is a quadruple

$$(pre, post, mod, term)$$

that consists of

- a precondition $pre \in \text{DLFml}$,
- a postcondition $post \in \text{DLFml}$,
- a modifier set $mod \in \text{Trm}_{LocSet} \cup \{\text{STRICTLYNOTHING}\}$,
- and a termination witness $term \in \text{Trm}_{Any} \cup \{\text{PARTIAL}\}$.

All contract components may refer to the special program variables `self` (unless `m` is static), `heap` and to the program variables $\mathtt{p}_i$ ($1 \le i \le n$) representing the method parameters. The postcondition may additionally refer to the program variables $\mathtt{heap}^{pre}$, `exc` and `res` (if the result type R of `m` is not `void`).

The postcondition can access more program variables, because it talks about two program states (before and after the execution) while the other components of the contract are all evaluated in a single program state—the state before the execution.

The modifier set *mod* deviates a little from the other components since it may be either a term (describing the set of locations that may be changed) or the string STRICTLYNOTHING which does *not* stand for a term but is an indicator subject to special treatment when proving and applying the contract. The set *mod* denotes the set of *existing* memory locations that m may modify; hence, the empty location set corresponds to `assignable \nothing`. A method with assignable STRICTLY-NOTHING must not change *any* location, not even a freshly created one; this fact can therefore not be expressed as a location set and requires the special indicator.

More on termination proofs for recursive methods can be found in Section 9.1.4, specifically in the rule in Definition 9.14.

Ghostbusters

When we speak of a contract for a method then, more precisely, we mean the complete program code consisting of the proper Java code and all JML annotations. That is not necessarily the same as the original Java code. Ideally, program code and its specification are strictly separated: only the proper program is executable, while its specification states a property on these executions. Unfortunately, that is not the case with *specification and annotation* languages like JML. In addition to contracts and invariants, JML has annotations that are placed as additional specification-only statements inside the code. These are assignments to ghost variables (i.e., `set` statements) or assertions (see Sections 7.7.2 and 7.9.3, respectively).

If contracts refer to the annotated code, then how can they make statements about the *original* program? The principal idea is that a program augmentation with JML statements must be a conservative extension w.r.t. program semantics, i.e., JML statements must not have effects on the part of the state space accessible by the Java program. Otherwise, the program executions which are considered during verification would be different from the ones actually run by a Java virtual machine, and the proofs worthless. The JML language rules forbid `set` statements to assign to regular Java locations. However, `set` statements (and even assertions) may still alter the control flow by raising exceptions. The absence of such exceptions—typically runtime exceptions—needs to be proven separately, see [Filliâtre et al., 2014].[18]

A dependency contract has fewer items:

Definition 8.3 (Dependency contract). A JavaDL method dependency contract (pre, var, dep) for a method consists of

- a precondition $pre \in \mathrm{DLFml}$,
- a termination witness $term \in \mathrm{Trm}_{Any}$,
- and a dependency set $dep \in \mathrm{Trm}_{LocSet}$.

All components may refer to the program variables `self` (unless `m` is static), `heap` and to the program variables $\mathtt{p}_i$ representing the method parameters.

[18] In KeY, not all checks have been implemented yet that are required to ensure JML statements are conservative extension.

The preprocessing of contracts within JML laid out in Section 8.2.1 was designed to provide a quite direct translation into JavaDL contracts: Every normalized functional JML contract that adheres to the template in Listing 8.1 becomes a functional contract according to Definition 8.2, while every JML dependency contract adhering to Listing 8.2 becomes a dependency contract according to Definition 8.3. The elements of the contracts are extracted from their JML counterparts as follows:

- The precondition *pre* and the dependency set *dep* of a JavaDL contract are the direct translation of their JML counterpart: $pre := \lfloor Pre \rfloor$, $dep := \lfloor Acc \rfloor$.
- The JavaDL postcondition combines the postcondition for normal termination *Post* and the exceptional termination postcondition *Signals* into one formula: $post := (exc \doteq \texttt{null} \rightarrow \lfloor Post \rfloor) \land (exc \not\doteq \texttt{null} \rightarrow \lfloor Signals \rfloor)$
- A special case exists for the modifier set. For most assignable clauses the modifier set *mod* is the JavaDL correspondent to the location set *Ass* specified as assignable clause. If the special symbol STRICTLYNOTHING has been used as assignable clause, however, the modifier set keeps this special symbol:
$$mod := \begin{cases} \textsc{strictlyNothing} & \text{if } Ass = \textsc{strictlyNothing} \\ \lfloor Ass \rfloor & \text{otherwise} \end{cases}$$
In Definition 8.4 and in equation (8.4) we will see that this case is treated specially in the construction of a proof obligation.
- The normalized JML contract allows only **`true`** or **`false`** as divergence clauses. The termination indicator *term* can be directly taken from the JML specification:
$$term := \begin{cases} \textsc{partial} & \text{if } Diverges = \texttt{true} \\ \lfloor Var \rfloor & \text{if } Diverges = \texttt{false} \end{cases}$$

The Use of Contracts

In this chapter, a correctness proof for a method contract stands on its own. Whenever a contract has been proved sound, it has been ensured that the formal requirement laid out in the specification is fully met by the implementation. The use of method contracts as abstraction of method invocation has already been briefly covered in Section 3.7.1, but it is only in the next chapter on modular specification verification that we will learn how method contracts can be used to reason about Java programs in a *modular* fashion. There, the contracts give rise to new calculus rules applicable to method calls in programs. Those rules are only sound if the corresponding proof obligations have been discharged. They go hand in glove like *lemmas* in mathematical proof tradition: the claim of a lemma corresponds to the specification, its proof corresponds to the proofs conducted in this chapter, and using it within another proof corresponds to applying calculus rules that will be introduced in the next chapter.

8.2.5 Loop Specifications

Methods and model fields are not the only syntactical constructs that can be annotated with a specification. Loops can also be furnished with a formal specification. Definition 3.23 introduced loop specifications as a triple $(inv, mod, term)$ of loop invariant, modifier set and termination witness. Section 7.9.2 has already outlined the syntax for loop specifications in JML:

```
/*@ maintaining maint;
  @ decreasing decr;
  @ assignable ass;
  @*/
```

JML allows the annotation of several loop invariants in one loop specification. If more than one loop invariant clause is given, the clauses are combined into one using `&&`. Table 8.3 lists the clauses allowed in loop specifications and their default values in case they are omitted. JML has synonyms for the loop specification keywords which are also listed in the table.

Table 8.3 Clauses in JML loop specifications

JML keyword	synonyms	default value
`maintaining`	`maintains`, `loop_invariant`	`true`
`decreasing`	`decreases`	PARTIAL
`assignable`		`\everything`

The translation from a JML loop specification as above into a JavaDL loop specification $(inv, mod, term)$ is straightforward and works as follows:

$$\begin{aligned} inv &= \lfloor maint \rfloor \\ mod &= \lfloor ass \rfloor \\ term &= \lfloor decr \rfloor \end{aligned}$$

The translation of JML expressions in loop invariants that make use of the `\old` operator requires a little attention: The old state refers to the state in which the enclosing method has been invoked; it does *not* refer to the state directly prior to loop entry, and it does *not* refer to the state after the last iteration.

The translation of heap expressions in `\old` refers to the heap variable $\mathtt{heap}^{pre}$ which has then been set to the according heap at method entry, a method parameter `p` is mapped to special purpose program variable $\mathtt{p}^{pre}$ in which `p`'s value at method entry is stored. Local variables are not affected by `\old`.

8.3 Proof Obligations for JavaDL Contracts

JML and JavaDL method contracts capture requirements on the behavior of Java methods in a formal manner. For the verification of method implementations, formulas will be introduced in the following that encode their correctness into JavaDL. Their validity is equivalent[19] to the correctness of the method implementation with respect to the contract of the method. On the other hand, if the formula can be falsified, the counterexample is a proof that the contract is not correct. Proof obligations thus define a semantics for JavaDL contracts: A method implementation fulfills its formal contract if and only if the corresponding JavaDL proof obligation is universally valid.

While proof obligations for methods can already be used to prove programs correct, the specification and verification of individual methods of a Java program is part of a greater task: the modularization of verification process. In Section 9.4.3 we will encounter inference rules that replace method invocations by instances of their JavaDL method contracts. These rules tie in with the proof obligations presented here in the sense that the correctness of the latter imply the soundness of the former rules presented in Chapter 9.

By the way: A method contract relevant for a method needs not be annotated with the method implementation under verification: Recall that JML features inheritance of contracts in order to implement the concept of behavioral subtyping (see Section 7.4.5). Therefore, if a method implementation overrides an implementation from a superclass or if it implements a signature declared in an interface, the implementation inherits all (nonprivate) specifications from the supertype.

8.3.1 *Proof Obligations for Functional Correctness*

Below we define a JavaDL formula whose validity is equivalent to the correctness of a function method contract. Unlike in other verification frameworks (e.g., earlier versions of KeY [Beckert et al., 2007]), we do not encode the verification condition into various assertions to be proved, but construct one single formula per contract. The general idea of this proof obligation is to show that the precondition implies that the postcondition holds after the execution of the method. But the postcondition is not the only guarantee that we are interested in: the assignable clause specified in JML (respectively the *mod* set in the JavaDL functional contract) states the locations that may be modified by the method; and this also needs to be checked. If $Contract = (pre, post, mod, term)$ with $term \neq \textsc{partial}$ is a functional contract for total correctness of method `m` according to Definition 8.2, both proof objectives can be expressed together as the formula

$$pre \rightarrow \langle \texttt{res = self.m(p}_1\texttt{,...,p}_n\texttt{);} \rangle post \wedge frame \tag{8.1}$$

[19] As a matter of fact, these formulas actually *define* the notion of correctness in KeY.

in JavaDL. The formula *frame* capturing the framing condition will be defined in (8.4) and (8.5) below. In Section 9.5, a concrete example for a functional contract proof obligation is examined more closely.

Free Preconditions

Since Java is a real-world programming language whose rich feature set has to be modeled logically in JavaDL, the above proof obligation is too simple. A number of adaptations need to be made to the proof obligation (8.1) to accommodate the idiosyncrasies of the Java language and its encoding in JavaDL. One point is that (8.1) is too strong since the initial state is only constrained by the precondition. The state space that JavaDL spans for all possibly definable interpretations of the logical symbols contains a lot more states than are reachable by the execution of Java programs. This includes the range of values that are admissible for programs. A typical example is that the `this` pointer (i.e., the program variable `self`) must not hold the `null` reference. From the logic's perspective, nothing speaks against this particular value; it must be ruled out explicitly: Additional assumptions must be made that constrain the states to those that can actually be reached by a Java program, thus weakening the proof obligation, e.g., by assuming $\mathtt{self} \not\doteq \mathtt{null}$.

This weakening improves precision of the proof obligation, yet it does not compromise its correctness since we are only interested in proving the contract correct w.r.t. all states reachable by a Java program. This additional assumption is called the *free precondition*:

$$\begin{aligned} \mathit{freePre} := \quad & \mathit{wellFormed}(\mathit{heap}) \\ & \land \mathtt{self} \not\doteq \mathtt{null} \\ & \land \mathtt{self}.\mathit{created} \doteq \mathit{TRUE} \\ & \land \mathit{exactInstance}_C(\mathtt{self}) \\ & \land \mathit{paramsInRange} \end{aligned} \tag{8.2}$$

The free precondition contains the assumption that the heap is well-formed (e.g., there are no dangling references, see Figure 2.7 in Section 2.4.3 on page 42), that the receiver object is of exact type C, and that the values of all parameters are within the bounds defined by their type:

$$\mathit{paramsInRange} := \bigwedge_{i=1}^{n} \begin{cases} p_i \doteq \mathtt{null} \lor p_i.\mathit{created} \doteq \mathit{TRUE} & \\ & \text{if the parameter is of reference type} \\ \mathit{inInt}(p_i) & \text{if the parameter is declared } \mathtt{int}\ p_i \\ \mathit{inByte}(p_i) & \text{if the parameter is declared } \mathtt{byte}\ p_i \\ \vdots & \text{likewise for } \mathtt{short}, \mathtt{long}, \mathtt{char} \\ \mathit{true} & \text{otherwise} \end{cases} \tag{8.3}$$

The predicates *inInt*, etc., are true if the argument is within the bounds of that type (`int` for *inInt*). See Section 5.4.3 for the semantics of the predicates in the various integer semantics available in KeY.

Also method $\mathtt{m(p_1,\ldots,p_n)}$ in (8.1) is subject to a change: The method call needs to be wrapped in a try-catch statement to capture an exception that might be thrown during the execution of `m` in the dedicated program variable `exc`—thus making thrown exceptions accessible to the postcondition. In the method call, it is also made specific which implementation of the method is to be used (dynamic binding is switched off) by using the method body statement (see Section 3.6.5) instead of the method call. Finally, an update is added to provide access to values from before the method execution.

Definition 8.4 (Proof obligation for functional contracts). Consider a functional method contract $Contract = (pre, post, mod, term)$ for the method $m(p_1,\ldots,p_n)$ declared in class or interface C. The implementation of m in a class $C' \sqsubseteq C$ is called correct with respect to *Contract* if the following JavaDL formula, called the *contract proof obligation* for *Contract*,

$$pre \land freePre \rightarrow \{\mathtt{heap}^{pre} := \mathtt{heap} \| \mathtt{exc} := \mathtt{null} \| \mathtt{mby} := term\}$$
$$\left[\!\!\left[\begin{array}{l}\mathtt{try\ \{\ res=self.m(p_1,\ldots,p_n)@C';\ \}} \\ \mathtt{catch(Throwable\ e)\ \{\ exc\ =\ e;\ \}}\end{array} \right]\!\!\right] (post \land frame)$$

is valid. The modality $[\![\cdot]\!]$ is instantiated by $[\cdot]$ if $term = \textsc{partial}$ and by $\langle\cdot\rangle$ otherwise. The assignment to `res` is omitted if m is declared `void`. The update $\mathtt{mby} := term$ is left out if $term = \textsc{partial}$.

This definition makes use of a formula *frame* (called the *framing condition*) encoding the proof obligation that the method does not change locations outside the modifier set *mod*. If $mod = \textsc{strictlyNothing}$, then the framing condition is

$$frame := \forall o \forall f;\ o.f \doteq o.f@\mathtt{heap}^{pre} \tag{8.4}$$

requiring that every location on the heap that is reached after the method invocation holds the same value as before that invocation. If *mod* differs from STRICTLYNOTHING, the condition is more sophisticated and reads as follows:

$$\begin{aligned} frame := \forall o \forall f;\quad & o.created@\mathtt{heap}^{pre} \doteq FALSE \\ & \lor o.f \doteq o.f@\mathtt{heap}^{pre} \\ & \lor (o,f) \in \{\mathtt{heap} := \mathtt{heap}^{pre}\}mod \end{aligned} \tag{8.5}$$

This condition states that any heap location (o,f) either

- belongs to an object o which has not (yet) been created before the method invocation, or
- holds the same value after the invocation as before the invocation, or
- belongs to the modifier set described by *mod* (evaluated in the prestate).

The framing problem will be topic of a larger discussion in Section 9.3.

The modality in the contract proof obligation is prefixed with an update which prepares a few program variables:

$\mathtt{heap}^{pre} := \mathtt{heap}$ the heap state before the method execution is stored in the program variable $\mathtt{heap}^{pre}$ to have it available for evaluation of the postcondition.

`exc` := `null` There is no exception observed initially. Unless an exception is raised, this variable will remain `null`.

`mby` := *term* The value of the termination witness at the beginning of the method is stored in *mby*. In Definition 9.18 in Section 9.4.3, we will see that when invoking a method n, its variant expression $term_n$ must be proved smaller[20] than `mby` to guarantee that there is no infinite recursion.

'Assignable' Semantics Versus 'Modifies' Semantics

There is a subtle difference in the understanding of **`assignable`** clauses between what JML defines and how KeY implements it in form of the proof obligation from Definition 8.4. In standard JML, a heap location may only ever be assigned to if it is contained in the assignable clause. That means that it must not occur on the left hand side of any Java assignment operator unless included in the assignable set (hence the name 'assignable').

KeY's dialect of JML, however, sees this a little more liberal: The assignable clause specifies the set of locations that may have a modified content after the method has finished. This semantics considers a location unchanged if it has the original value at the end of the method call. It may change its value throughout the course of the method as long it regains the old value at the end of the method. We called this set the 'modifies' set for that reason. It is evident that the assignable semantics is stricter than the modifies semantics. Every program that is correct with respect to former is correct with respect to the latter.

The opposite direction does not hold. Listing 8.5 shows a small example of a program that is correct w.r.t. modifies semantics, but not w.r.t. to the assignable semantics: The method must not 'change' any existing location on the heap (**`assignable \nothing;`**). The value of **`this`**`.f` is temporarily changed in line 9 but restored directly afterwards to the original value such that at the end of the method, the original value is in **`this`**`.f` again (at least in sequential, single-threaded programs). For sequential programs, in which only the initial and terminal state are relevant, nothing speaks against the more liberal understanding since intermediate violations cannot be observed. The 'modifies' semantics admits more programs in which locations may have their values changed during the run. The choice of semantics does make a difference, however, for multithreaded

[20] w.r.t. a well-founded ordering

programs where threads may rely upon the fact that a parallelly executed thread keeps the heap state untainted.

```
class Assignable {
   int f;

   /*@ normal_behavior
     @  assignable \nothing;
     @*/
   void pureMethod() {
      int old = this.f;
      this.f = 0;
      this.f = old;
   }
}
```

Listing 8.5 The two semantics of `assignable` clauses

The general proof obligation as it has been introduced in Definition 8.4 applies to normal Java methods. The general idea for the proof obligation applies as well to special types of methods and to model methods. However, there are cases that differ from the above pattern and we will in the following list the proof obligations for constructors, abstract classes and model methods.

8.3.1.1 Constructors

The proof obligations for constructor contracts are a little different from the proof obligations presented in Definition 8.4 for ordinary Java methods. It is the Java block within the modality which has to be modified; for a constructor $A(T_1\ p_1, \dots, T_n\ p_n)$ for a class A it reads:

```
try {
  A self = A.<createObject>();
  self.<init>(p1, ..., p_n);
  self.<initialized> = true;
} catch(Throwable e) { exc = e; }
```

The Java code fragment makes use of the synthetic methods `<createObject>` and `<init>` and the synthetic Boolean field `<initialized>` which are not part of the Java language but additions introduced in the context of symbolic execution and object creation in KeY. See Section 3.6.6.3 for an introduction to these synthetic symbols and on how they are used during symbolic execution of object creation.

Note that the contract for a constructor does not only span over the initializing code in the constructor's body but also includes the creation of the object. This has as an implication that the `this` reference (which points to a created object after the constructor) is a not-yet-created object in the prestate: `\fresh(this)` is a valid postcondition for any constructor.

8.3.1.2 Methods in Abstract Classes

If the class C is declared `abstract`, there cannot be objects that are exactly of that type. The predicate $\mathit{exactInstance}_C(\mathtt{self})$ can thus never hold, the free precondition is always false, and the condition in Definition 8.4 is trivially valid. This seems against the semantics of method contracts, but since proof obligations exist also for inherited methods, it is ensured that every running implementation is verified against their contracts.

The KeY system treats abstract classes specially in that it suppresses the creation of the corresponding trivial proof obligations altogether to allow the user to focus on the relevant proof obligations.

8.3.1.3 Model Methods

For a strictly pure model method with a single side-effect-free return statement, the Java modality can be replaced by an update. Let the body of a model method be `return` *Expr* for some JML expression *Expr*.

The proof obligation for a such model method is thus

$$\mathit{pre} \wedge \mathit{freePre} \rightarrow \{\mathtt{exc} := \mathtt{null} \| \mathtt{mby} := \mathit{term}\}\{\mathtt{res} := \lfloor \mathit{Expr} \rfloor\}\mathit{post} \enspace .$$

in which a simple update takes the place of the Java modality.

Since *Expr* is side-effect-free, exceptions need not be considered here. An advantage of this formulation of the model method proof obligation is that JML-expressions (going beyond the Java language) can be used in the return statement as they need not go through symbolic execution.

For model methods which are not strictly pure or which have a nontrivial method body, a modality like in Definition 8.4 must be used. If the body additionally makes use of JML-only expressions or statements, a more liberal modality operator which allows for JML constructs in Java programs is needed. Currently, this is not supported in the KeY system.

8.3.1.4 Static Methods

Static methods differ from instance methods essentially in one respect: They do not have a “`this`” reference pointing to receiver object. For a static method the proof obligation of Definition 8.4 therefore needs to be adapted by

1. dropping the conjuncts in the free precondition *freePre* which refer to the program variable `self` and by
2. changing the method body statement such that it refers to the class rather than to the receiver object self. (The assignment in the modality then reads $\mathtt{res} = \mathtt{C.m}(\mathtt{p_1},\ldots,\mathtt{p_n})@C$.)

8.3.2 Dependency Proof Obligations

We are not solely interested in verifying that the result of a method invocation adheres to a given postcondition, but we are also interested in formalizing, specifying and verifying that the result of a method depends at most on a given set of locations on the heap.

This question is closely related to the noninterference problem examined in the light of information flow properties in Chapter 13, and dependency checking can be regarded as a special case of noninterference checking.

In Section 8.3.1, we discussed that for assignable clauses we do not check that every write operation affects a location in the set of modifiable locations, but rather look at the locations' contents in the end. A similar situation arises now for checking read accesses to heap locations. One approach would be to check (by adding assertions during symbolic execution) that every read access is to an admissible location.

Like for checking assignable clauses, we take a more liberal approach that requires checking an assertion only after the execution of the method has finished: We assert that the result of the method is *semantically* independent from all locations from which it must not read. This is more liberal than read access checking in that it allows a location to be read as long as the value does not have any influence on the method's result. In the expression `o.f*0`, for instance, it not necessary that `o.f` is in the dependency set since the result of the operation is constant and does not depend on the location's value though that occurs syntactically in the evaluation.

Now, the task is to come up with a JavaDL proof obligation for this independence. One technique to formalize that the result of a method m depends at most on a set of inputs specified in *dep* is to prove the following: Invoking the method in two memory states that agree on memory locations in *dep* (but may disagree on all other locations) must yield the same result. This formalization of noninterference is called *self-composition* (see [Darvas et al., 2003, 2005]), and a variation of it is also used for noninterference proofs with KeY, see Section 13.5.1 for details.

Definition 8.5 (Proof obligation for dependency contracts). Consider a method dependency contract $Contract = (pre, term, dep)$ for the method $T\ \mathtt{m}(\mathtt{p}_1, \ldots, \mathtt{p}_n)$ declared in class C with $T \neq \mathtt{void}$. The implementation is called correct with respect to $Contract$ if the following JavaDL formula, called the *dependency contract proof obligation*,

$$
\begin{aligned}
&\quad \mathit{pre} \land \mathit{freePre} \land \mathit{wellFormed}(h) \land \mathtt{mby} \doteq \mathit{term} \\
&\land \mathtt{heap}_2 \doteq \mathit{anon}(\mathtt{heap}, \mathit{setMinus}(\mathit{allLocs}, \mathit{dep}), h) \\
&\land \qquad\qquad\qquad\quad [\mathtt{res\ =\ self.m(p}_1\mathtt{,\ \ldots,\ p}_n\mathtt{)@C;}]\mathtt{res} \doteq r_1 \\
&\land \{\mathtt{heap} := \mathtt{heap}_2\}[\mathtt{res\ =\ self.m(p}_1\mathtt{,\ \ldots,\ p}_n\mathtt{)@C;}]\mathtt{res} \doteq r_2 \\
\to&\quad r_1 \doteq r_2
\end{aligned}
$$

is valid. In this formula additional constants $h, \mathtt{heap}_2 : \mathit{Heap}$ and $r_1, r_2 : \lfloor T \rfloor$ are used.

The rule is implemented slightly differently (yet equivalently) in the KeY system where the proof obligation coincides with the one for dependency contracts for general observer symbols as introduced in Definition 9.12 in Section 9.3.3.

Our interpretation of accessible clauses requires only that the result value of a method must depend at most on the locations in *dep* while any heap location may be modified without restriction. This deviates from the semantics defined for JML where every effect (on result or heap state) may depend at most on the part of the heap specified in a `accessible` clause.

In the course of Chapter 9 we will see that dependency contracts play an important role for modular reasoning within the KeY approach. Their primary use case is to specify in which cases pure methods used within specifications return the same result. For this purpose it is natural to only analyze dependency of the method return value disregarding all effects on heap locations. The reader who is interested in more accurate and general specification and more powerful verification of information flow properties using KeY is referred to Chapter 13.

8.3.3 Well-Definedness Proof Obligations

Some operators of the expression language have a canonical semantics only for a subset of possible inputs, and the meaning of expressions in which such operators are applied outside this set—called the operator's *domain*—is yet to be defined. As an example, consider the following method specification

— Java + JML —
```
/*@ public normal_behavior
  @  ensures \result >= 1000 / n;
  @*/
int m(int n) { ... }
```
— Java + JML —

which postulates that `1000/n` is a lower bound for the result value. If somewhere in the program the method is invoked via `m(0)`, the problem of the specification becomes apparent: To evaluate the postcondition, the expression `1000/0` would also have to be evaluated—but what is the result of this operation?

Since the expression language of JML is an extension of the side effect-free expressions in Java, it is desirable that the semantics of Java expressions should be retained if they are used in JML context. This is problematic since evaluating `1/0` in Java does not give a value but raises a `DivisionByZeroException`. Exception handling is a concept for managing program control flow and not for expression evaluation: If an exception is raised during expression evaluation, control flow is transferred abruptly and the according expression does not give any value.

In JavaDL, all functions and predicates are total such that every expression always yields a value in its co-domain. The expression $1/0$ evaluates to an integer value—however, we cannot assume anything about this value, except that it is an integer. This approach is called *underspecification*; see Section 2.3.2 and [Schmitt, 2011, Sect. 2] for more details. It has the advantages of being easily definable and that axioms of classical logic are still valid. If a function symbol is applied to argument values which are not in its domain, then the function symbol is left uninterpreted for these input values. For a formula to be valid, it is required to be satisfied for all possible results in the undefined places; i.e., it must be valid in all structures which lift the places of partiality with an arbitrary value. For example, two interpretations that map $1/0$ to 0 and $1/0$ to 42, respectively, both need to be taken into consideration when proving the validity of a formula. The property $\exists \mathit{int}\, x;\ x \doteq 1/0$ is valid since JavaDL's division is a total function and in any model there is one integer (albeit unknown) value which is equal to $1/0$. The equality $1/0 \doteq 1/0$ is also valid due to the reflexivity of $\doteq$. However, neither of the statements $1/0 > 0$ nor $\neg(1/0 > 0)$ is valid since $1/0$ is positive in some interpretations and is nonpositive in the others. Likewise, the equality $1/0 \doteq 2/0$ is neither valid nor unsatisfiable; it also depends on the semantics of the underspecified parts of integer division.

There are several concepts to model undefinedness logically. Besides underspecification, the issue of undefined function applications can be modeled using a dedicated error element, three-valued logics, dependent types, or partial functions to name a few concepts. For an extensive comparison refer to [Hähnle, 2005].

In the following, proof obligations will be introduced that show that an expression does not depend on the semantics of undefined function applications. Hence, the concept by which function applications outside domains are modeled becomes irrelevant since the valuation of expressions is guaranteed not to be influenced by external valuations.

The analysis presented in the following is targeted at JML specifications. This raises the question: What are the admissible argument values for the JML operators? According to the JML reference manual [Leavens et al., 2013], a Boolean expression is satisfied in a state if it has the truth value true and "does not cause an exception to be raised." Raising an exception is thus the Java/JML indication for applying a function outside its domain. We capture this in the following definition which cannot be entirely formal since we have not formally defined the concepts of memory state (being program execution contexts with local variables, heap, method call stack, ...).

Definition 8.6 (Well-definedness). Given a JML expression e and a memory state s, the expression e is called *well-behaving* in s if the from-left-to-right short-circuit evaluation of e in s does not raise an exception. The expression e is called *well-defined* if it is well-behaving in all memory states.

A JML method contract is called well-defined if

1. its precondition is well-defined, and
2. all clauses evaluated in the prestate are well-behaving in all states that satisfy the precondition, and
3. all clauses evaluated in the poststate (i.e., `signals` and `ensures` clauses) are well-behaving in all states which are reachable by executing the method in a state satisfying the precondition.

The intuition behind this exception-based definition becomes more natural when considering another use case of JML specifications (besides deductive verification): During runtime assertion checking, a specification is to be refused and to be considered ill-defined if its evaluation causes an exception.

When checking JML contracts, it is always the precondition which is checked before anything else. Hence, all other specification elements are only ever evaluated if the precondition holds (and is well-behaving). Hence for the well-definedness of contracts, the fact that the precondition holds[21] can be safely assumed when investigating the well-definedness of other specification elements. We say that the precondition *guards* the other specification elements. The idea of a conservative formulation of specifications in which the precondition guards the postcondition has been brought forward by Leavens and Wing [1998].

Example 8.7. The method contract for method `m` introduced at the beginning of the section is not well-defined since there is a memory state (namely, if $n = 0$) in which the postcondition is not well-behaving.

The situation can be remedied by adding the precondition `requires n != 0;` to the contract. Under assumption of this precondition, the division `1000/x` does not raise an exception; the postcondition is well-defined.

It is not only the precondition that guards other parts of a specification. Since expressions are evaluated from left to right in Java (and, hence, also in JML), it is possible to guard subexpressions which occur 'further to the right' from within the same specification element. As soon as the result of certain Boolean operations is inevitable in the evaluation of a Java expression, the remainder of the expression is no longer considered for evaluation. This is called *short-circuit* or *lazy* evaluation. When the JVM computes the value of A `&&` B for the short-circuited conjunction `&&`, it first evaluates A and if that is false, the conjunction is falsified and B needs not be evaluated for the result. In the logic we can formulate this as

$$val(\lfloor A \texttt{ \&\& } B \rfloor) = \begin{cases} \mathit{ff} & \text{if } val(\lfloor A \rfloor) = \mathit{ff} \\ val(\lfloor B \rfloor) & \text{if } val(\lfloor A \rfloor) = \mathit{tt} \end{cases}$$

[21] *nota bene*: we assume that the precondition holds, not that it is well-behaving or well-defined.

The value $val(\lfloor B \rfloor)$ is only referred to if A evaluates to true. Hence, It suffices when B is well-behaving in states where A is satisfied. In classical (two-valued) logics, this definition is not different from the usual definition of conjunction. It *is* different if one allows *val* to fail, for instance, by giving a special error truth value different from *tt* and *ff*.

Example 8.8. The class invariants of the following class are well-defined since all possibly not well-behaving operations are guarded.

Java + JML

```
1 class GuardExample {
2   int[] values;
3   int length;
4   /*@ nullable @*/ GuardExample next;
5
6   //@ invariant next != null ==> length == next.length+1;
7   //@ invariant (\forall int i;
8   //@    0<=i && i < values.length; values[i] > 0);
9 }
```

Java + JML

The first invariant in line 6 aligns the values of the `length` fields of nodes in a singly linked list. In the expression `next.length`, the operand `next` may be `null` and the field access locally ill-behaving. But when evaluating the entire invariant, it cannot raise an exception since the implication operator `==>` has short-circuit semantics. If `this.next` is different from the `null` reference, the equality can be evaluated without raising an exception; and if `this.next` is `null`, then the left hand side of the implication evaluates to false and the expression is already true without evaluation of the equality.

The second invariant in line 7 is also well-defined since for every possible value for `i`, the range check `0<=i && i<values.length` guards the array access `values[i]` which can thus not cause an `ArrayIndexOutOfBoundsException`.

8.3.3.1 Well-Definedness of JML Expressions

In order to be able to describe the proof obligations which come up for JML expressions, we introduce a new transformation function ω which takes a JML expression and produces a JavaDL formula from it.

Definition 8.9. The well-definedness term transformation operator $\omega : \text{JExp} \to \text{DLFml}$ assigns to every JML expression a JavaDL formula. It is defined in Appendix A.3. For its evaluation, it makes use of the translation function $\lfloor \cdot \rfloor$ from Definition 8.1.

The intention behind ω is that whenever $\omega(e)$ is true then e is well-behaving. This logical notion of a well-definedness condition refines the informal Definition 8.6 and leads to

Proposition 8.10. *Let s be a memory state and $\lfloor s \rfloor$ the corresponding JavaDL structure. and $e \in \mathrm{JExp}$. If $\lfloor s \rfloor \models \omega(e)$, then e is well-behaving in s. If $\omega(e)$ is universally valid, then e is well-defined.*

A formal proof is omitted mainly since it would require the formalization of left-to-right short-circuit evaluation for the entire JML language, which we do not want to provide here.

Instead, we focus on central items of ω's definition, the full definition can be found in Appendix A.3. For many JML function and operator applications, well-definedness of the application reduces to well-definedness of all arguments. Only if the function's domain is restricted, additional requirements are to be met. For the arithmetic expressions we have, for instance,

$$\begin{aligned} &\omega(A \texttt{ + } B) = \omega(A) \wedge \omega(B) \qquad \text{(accordingly for } \texttt{*, +, -, <, ==}, \dots) \\ \text{but} \quad &\omega(A \texttt{ / } B) = \omega(A) \wedge \omega(B) \wedge \lfloor B \rfloor \not\doteq 0 \qquad \text{(accordingly for \texttt{\%})} \ . \end{aligned}$$

Note that the well-definedness transformation ω refers to the evaluation transformation $\lfloor \cdot \rfloor$. Boolean expressions support short-circuit evaluation as mentioned above:

$$\begin{aligned} \omega(A \texttt{ \&\& } B) &= \omega(A) \wedge (\lfloor A \rfloor \rightarrow \omega(B)) \\ \omega(A \texttt{ ==> } B) &= \omega(A) \wedge (\lfloor A \rfloor \rightarrow \omega(B)) \\ \omega(A \texttt{ || } B) &= \omega(A) \wedge (\neg \lfloor A \rfloor \rightarrow \omega(B)) \\ \text{but} \quad \omega(A \texttt{ \& } B) &= \omega(A) \wedge \omega(B) \end{aligned}$$

An interesting question for short-circuit evaluation is the extension of the concept to quantifiers. One can argue that the existentially quantified formula `(\exists int x; 1/(x+1) == 1)` is well-defined since there exists a witness (the value 0) that makes the statement true such that all other evaluations are irrelevant and can be omitted due to short-circuit evaluation. That would require, however, that in a left-to-right evaluation the matrix of the quantifier is evaluated at 0 before it is evaluated at -1 (which would raise an exception). To do so, a (well-)ordering of the values of the quantified domain must be fixed such that in the sequence of valuations those coming later are guarded by those ordered before. In the case of integers, a natural order might be suggested, but for other domains (like `Object`), no canonical, intuitive order comes to mind. For that reason, we opt for a conservative approach for the well-definedness of quantifiers: The valuation of one instantiation cannot guard another instantiation, and we have

$$\omega(\texttt{(}\backslash Q\ T\ v\texttt{;}\ A\texttt{;}\ B\texttt{)}) = \forall \lfloor T \rfloor\, v;\ \big(\omega(A) \wedge (\lfloor A \rfloor \rightarrow \omega(B))\big)$$

for a generalized quantifier $Q \in \{\texttt{forall}, \texttt{exists}, \texttt{sum}, \texttt{infinte\_union}, \texttt{product}, \texttt{min}, \texttt{max}\}$, where a missing guard A defaults to `true` as usual.

Another important issue of well-definedness is `null` dereferencing. The operator o`.f` for a field access is only well-behaving if the receiver object is different form the `null` reference. The same applies to array accesses, where the index must additionally lie within the array bounds such that we have

$$\omega(A\,.\,\mathtt{f}) = \omega(A) \wedge \lfloor A \rfloor \not\doteq \mathtt{null} \qquad \text{for a field access}$$

$$\text{and} \quad \omega(A\,[B]\,) = \omega(A) \wedge \omega(B) \wedge \lfloor A \rfloor \not\doteq \mathtt{null} \wedge 0 \leq \lfloor B \rfloor \wedge \lfloor B \rfloor < \mathit{length}(\lfloor A \rfloor)$$

for an array access.

For references to method invocations within specifications, the design-by-contract principle persists: A method invocation must satisfy the precondition of the contract. Let $\mathtt{Pre}_m$ be the functional precondition[22] of method $\mathtt{m}$ (compare Listing 8.1). The precondition refers to the formal receiver **`this`** and parameters $\mathtt{p}_1, \ldots, \mathtt{p}_n$ which have to be replaced by the concrete receiver o and the arguments a_i:

$$\omega(o\,.\,\mathtt{m}(a_1, \ldots, a_n)\,) = \omega(o) \wedge \lfloor o \rfloor \not\doteq \mathtt{null} \wedge \bigwedge_{i=1}^{n} \omega(a_i) \wedge \left\lfloor \mathtt{Pre}_m[\mathtt{p}_1/a_1, \ldots, \mathtt{p}_n/a_n, \mathtt{this}/o] \right\rfloor$$

8.3.3.2 Well-Definedness of Method Contracts

Using the well-definedness term transformation ω, we can now also define a condition for the well-definedness of method contracts. According to Definition 8.6, well-definedness of contract clauses other than the precondition is only required conditionally under assumption of the precondition being satisfied (the precondition guards the other clauses).

Definition 8.11 (Method contract well-definedness). Let the normalized method contract according to Listing 8.1 for method $\mathtt{m}$ be given. The well-definedness proof obligation for the contract for $\mathtt{m}$ is

$$\begin{aligned}\omega(\mathtt{m}) = \omega(\mathit{Pre}) \wedge &\\ (\lfloor \mathit{Pre} \rfloor \to \omega(\mathit{Var}) \wedge \omega(\mathit{Ass}) \wedge &\\ \{\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \lfloor \mathit{Ass} \rfloor, \mathtt{heap}') \| \mathtt{heap}^{\mathit{pre}} := \mathtt{heap}\} &\\ (\omega(\mathit{Post}) \wedge (\lfloor e \rfloor \not\doteq \mathtt{null} \to \omega(\mathit{ExPost})))) &\end{aligned}$$

and the well-definedness proof obligation for the dependency contract in Listing 8.2 reads

$$\omega(\mathtt{m}) = \omega(\mathit{Pre}) \wedge (\lfloor \mathit{Pre} \rfloor \to \omega(\mathit{Var}) \wedge \omega(\mathit{Acc})) \ .$$

While the precondition *Pre*, the variant *Var*, the accessible clause *Acc*, and the assignable clause *Ass* are all evaluated in the prestate of the method invocation, the postcondition *Post* and the exceptional postcondition *ExPost* are evaluated in the poststate. Since the specification should be consistent in itself without reference to the implementation, the poststate should not be considered as the state after the

[22] If $\mathtt{m}$ has more than one contract, it suffices to satisfy one of the preconditions. Then $\mathtt{Pre}_m$ refers to the disjunction of the preconditions of all functional method contracts which guarantee absence of exceptions (e.g., by being annotated `normal_behavior`).

execution of the implementation. Instead a most general poststate is assumed in which all assignable locations on the heap are assigned an unknown value using the anonymizing function *anon* (see Figure 2.11 in Section 2.4.5).

For well-definedness, the order of clauses is important. During normalization, if several clauses of a kind are present, they are combined into one (see Section 8.2.1.5). Two preconditions are conjoined into one using the short-circuit conjunction `&&` such that preconditions mentioned earlier in a contract guard preconditions which are mentioned afterwards. Nonnullness among other implicit assumptions is made explicit during normalization (see Section 8.2.1.2). The explicit clauses which result from this normalization are added *before* the explicit preconditions such that they can guard them.

Example 8.12. Consider the following method contract with two preconditions.

—— Java + JML ————————————————————

```
//@ requires a.length > 0;
//@ requires a[0] == 0;
void m(/*@non_null*/ int[] a);
```

———————————————————— Java + JML ——

A desugaring normalization of this contract results in the following contract with a single precondition

—— Java + JML ————————————————————

```
//@ requires a != null && a.length > 0 && a[0] == 0;
void m(/*@nullable*/ int[] a);
```

———————————————————— Java + JML ——

in which `a!=null` and `a.length > 0` together guard the array access `a[0]` such that this contract is well-defined.

8.3.3.3 Observations Concerning Well-Definedness

Modularity of Well-Definedness Proof Obligations

A remark on the precision of well-definedness for well-behaving specifications: It may be that a specification behaves well for all runs of a program, but that expressions—when inspected in isolation—are not well-defined. If `x` is a variable whose value is set to 1 initially, and never changed later, then the expression `1/x` is well-behaving within the context of this program. However, if the context (i.e., the concrete program) is removed and the expression is considered in isolation, `1/x` needs to be considered ill-defined. This is the view that KeY takes.

It is a deliberate choice: Specifications should be checked modularly for well-definedness ignoring the concrete program which they annotate. Well-definedness should *not* be a property depending on the behavior of the program. This has the benefit that if the program context changes (modification or extension of the program text), the well-definedness of the specification is not compromised.

Method contracts can always be made well-defined by adding the necessary assumptions guaranteed by the code explicitly to the specification. Making the guards explicit also clarifies specifications since it explicitly points out corner cases which are often the reason behind misunderstood specifications.

On The Evaluation Of Ill-Behaving Contracts

Up to this point, we have defined when a contract is or is not well-defined and found proof obligations to show its well-definedness. But, what happens if a clause of a contract is evaluated despite being ill-defined?

JML answers this by saying that any ill-behaving Boolean expression evaluates to false. This is called *strong validity* by Chalin and Rioux [2008]. It can be rephrased as 'an expression in which undefined subexpressions occur syntactically is not satisfied.' This is a very restrictive definition; occurrences of undefined expressions have an effect on satisfiability even if they do not matter in the classical logic sense.

Originally, JML propagated underspecification for the semantics of ill-defined expressions; the JML expression `1/0 == 1/0` would have been evaluated to true, now it evaluates to false. Chalin [2007] demonstrated through empirical studies that this semantics for function application outside of definition domain operations did not match programmers' expectations. For a software engineer, the expression `1/0` does not have a value but raises an exception which has to be dealt with.

Left-to-Right Versus Bidirectional Evaluation

JML defines the evaluation of expressions from left to right with short-circuiting as shown above. This is owed to the fact that JML semantics is based on the semantics of the Java programming language.

However, when discussing the issue of well-definedness of logical formulas, there is no reason why the formula $x \neq 0 \wedge 1/x > 0$ should be treated any differently from $1/x > 0 \wedge x \neq 0$—the conjunction is commutative after all.

And it *is* possible to define well-definedness symmetrically such that both conjuncts guard each other and the order of arguments to connectives does not play a role for their semantics. If done naively, the bi-directional guarding produces well-definedness proof obligations which are exponential in the length of the original formula. However, Darvas et al. [2008] proposed an efficient encoding that produces well-defined conditions with bidirectional guarding with linear effort.

However, it was not the complexity of the proof-obligation that was the rationale for the choice of JML semantics, but its heritage from Java semantics. The expression `1/x > 0 && x != 0` *may* throw an exception in Java even if its truth value does not depend on the value of `1/x`.

The well-definedness proof mechanism implemented in KeY (see [Kirsten, 2013]) supports both left-to-right and efficient bidirectional short-circuiting semantics.

Well-Definedness Is An Issue Of Pragmatics

Is well-definedness an issue of the syntax or the semantics of the specification language? There are aspects of well-formedness at the syntactic level: the adherence of programs and specifications to the languages' grammars and their well-typedness. The syntax and type checking for JML and JavaDL can be done efficiently on a syntactical level. However, well-definedness following Definition 8.6 is not a syntactic property that can be checked automatically by an efficient static analysis in all cases; it is not decidable. At the same time, it is wise to separate the concerns of well-definedness of a specification from its meaning. From a language designer's point of view, well-definedness is neither a syntactic nor a semantic problem but answers to the question whether a statement is sensible (in its context). Linguists call the field of interpretation of statements beyond that of its bare (model-theoretic) evaluation the *pragmatics* of a language. The statement $1/x > 0$, for instance, might invoke in the reader the *implicature* "x cannot be 0 since it occurs as denominator in a division." Doing well-definedness checking ensures that pragmatic issues outside the semantic truth value evaluation do not arise.

Auxiliary JML statements that may occur within the code (such as the set statement to assign to ghost variables/fields or loop invariants) have not been considered here, and neither have been class or loop invariants. For details on their treatment in KeY, see [Kirsten, 2013]. The implementation in KeY is slightly different from the current presententation, because it operates on JavaDL, where well-definedness of heap and field expressions is checked in addition.