Karlsruhe Institute of Technology

Translating Alloy Specifications to JML

Masters Thesis of

Daniel Grunwald

At the Department of Informatics
Institute for Theoretical Informatics (ITI)

Reviewer:	Prof. Dr. Mana Taghdiri
Advisor:	Dr. rer. nat. Christoph Gladisch
Second advisor:	Tianhai Liu, M. Sc.

Duration: April 7th 2013 – November 4th 2013

Ich versichere wahrheitsgemäß, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Institut für Technologie zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Karlsruhe, November 4th 2013

. . . .

Abstract

The lightweight Alloy specification language has been used for the specification of Java programs in the context of test-case generation and bounded verification. The aim of this thesis is the design of an automatic translation from Alloy into the Java Modeling Language (JML). Our tool "Alloy2JML" transforms relational Alloy formulas into JML expressions that represent relations using first-order logic. We use a translation function that has existential semantics and is defined recursively over the Alloy syntax tree. It is followed by a series of simplification steps that help remove redundant quantifiers introduced by the translation. The transitive closure operator is translated into recursively specified model methods.

The translation into JML allows the usage of JML tools on the input programs using Alloy specifications. We use the KeY theorem prover on the output of the translation to verify the correctness of two example programs.

The translation of a subset of the Alloy language is formally proven correct using the Isabelle theorem prover.

Keywords: Alloy, JML, Alloy2JML, KeY, Transitive Closure

Contents

1. Introduction

The Java Modeling Language (JML) [LPC$^+$11] is a behavioral interface specification language for Java. It combines Eiffel-style design-by-contract programming with Refinement Calculus. Contracts are embedded as annotations in the Java source code of the program, using a comment syntax that hides them from normal Java compilers. For the purpose of this work, we will concentrate on method specifications (preconditions, postconditions and frame conditions) and object invariants.

There are a wide variety of tools available for JML: a runtime assertion checker, static verification tools like ESC/Java2, OpenJML, LOOP, JACK; and test-case generation tools like jmlunit or JET. Of interest to our work is the KeY project [BHS07], which provides an interactive theorem prover with a JML front-end.

Alloy [Jac06] is a relational specification language. It is a popular language for lightweight modeling due to its simple semantics, conciseness and especially its fully automatic analyzer. An Alloy model consists of (among other items):

- type signatures, which define sets of atoms
- field signatures, which define relations
- facts, which are relational formulas that constrain the model

The Alloy Analyzer can be used to find and visualize instances of the Alloy model, or to verify an assertion about the model by looking for counterexamples and visualizing them. For this, the user has to provide bounds on the number of atoms in each type signature. The Analyzer works by translating the Alloy model into propositional logic and finding a model using a SAT-solver. Alloy is used in a wide variety of areas: from verifying distributed protocols like Chord[Zav12], over validation of security models, to test-case generation and program verification.

In the context of design-by-contract programming for Java, Alloy can be used for test-case generation using tools like TestEra [KYZ$^+$11] and FineFit [DGT13]. These tools automatically generate test cases that satisfy the method preconditions, which are specified using the Alloy language. Alloy can also be used for bounded verification of Java programs, for example using the JForge tool [DCJ06]. JForge transforms the Java program into Alloy's relational logic, and verifies the program against the Alloy specification within user-provided bounds on heap size and number of loop iterations.

To enable a full correctness proof for Java programs annotated with Alloy specifications, we introduce an automatic translation from Alloy to JML. This allows the use of JML tools on such specifications. In particular, we will use the KeY theorem prover to show the correctness of two example classes.

Why use Alloy and translate it to JML instead of directly writing JML specifications? One advantage is that using sets and relations is more expressive for many problems. In particular, contracts involving set of all nodes in a linked list or binary tree are much easier to specify using Alloy's transitive closure operator. Also, the relational override operator ++ can be useful for compactly specifying frame conditions.

Another reason exists when the system was previously modeled on a more abstract level using the Alloy Analyzer. In this case, it is useful to keep using the same language for the specification of the implementation, as it allows re-using and refining the existing specifications.

Finally, the use of Alloy as a specification language followed by a translation to JML allows using both Alloy and JML tools on the same program.

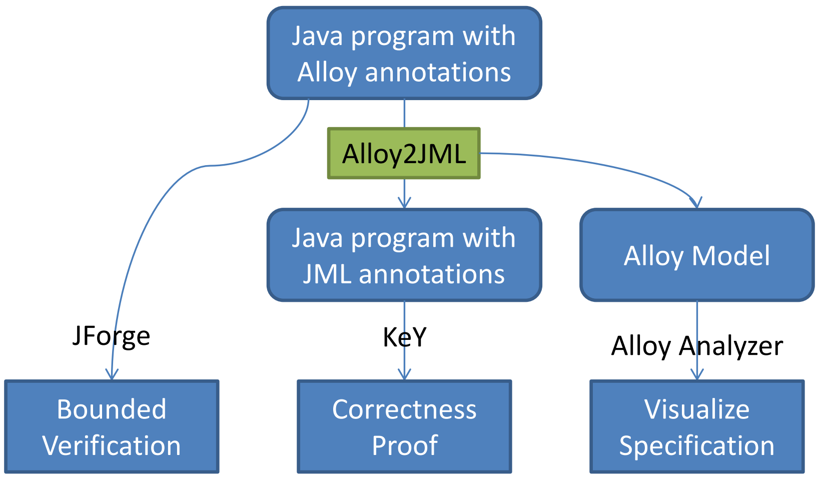

Figure 1.1.: Possible use-cases for Alloy2JML

Our translation software is called Alloy2JML. As input, it takes a Java program annotated with specifications in Alloy. The output of the translator is the same Java program, annotated with JML specifications. Additionally, our software outputs an Alloy model that contains Alloy signatures for the classes declared by the Java program, and Alloy signatures for the method specifications. This is very similar to the Alloy model generated by the TestEra. This model is used internally by our software for semantic analysis of the Alloy expressions used in the specification. A user may use this model to visualize her specification using the Alloy Analyzer. An overview of these possible use-cases is given in figure 1.1.

One possible use of the output Alloy model is to sanity-check invariants or preconditions: the Alloy Analyzer is used to visualize instances that satisfy the invariants and/or preconditions. By manually viewing several satisfying instances, the user can verify that all of these are indeed of the expected structure. Unexpected structures indicate that the invariants/preconditions are underspecified or incorrect. In this usage, it is often necessary to write additional predicates within the Alloy Analyzer to restrict the search space.

Another related use case is to visualize the postcondition for a method call with given arguments. If the Alloy Analyzer can find multiple instances for a method call with exactly specified input heap and arguments, this indicates that the postcondition does not exactly specify the result of the method. Such underspecification may be intentional, for example if the method allocates temporary helper objects without mentioning them in the contract. But it can also indicate unintentional underspecification. In one of our examples, we prove the `add(int value)` method of an unbalanced binary search tree. Using the Alloy Analyzer, we found that our first attempt at the method contract allowed the `add()` method to re-use existing nodes of an unrelated tree instead of creating a new node as expected.

In the next chapter, we will give a short overview of the Alloy and JML languages, to give the background needed to understand this thesis. In chapter 3, we describe the input format of our translation, i.e. how Alloy specifications are embedded in Java programs. In chapter 4, we show how these specifications can be visualized with the Alloy Analyzer. The translation from Alloy to JML using the translation functions $\mathcal{B}$ and $\mathcal{E}$ is explained in detail in chapter 5. Then, in chapter 6, we describe the translation of the transitive closure using JML model methods. After that, we proof the correctness of the translation in chapter 7 using the Isabelle/HOL theorem prover. At last, in chapter 8, we describe our experiences using the KeY theorem prover to prove the correctness of two sample programs with regards to a specification translated from Alloy. Finally, we conclude.

2. Background

2.1. Alloy

An Alloy model starts with a declaration of type signatures. The canonical example in introductions to Alloy specifies a file system:

```
// A file system object in the file system
abstract sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }
```

This example declares three type signatures: `FSObject`, `Dir` and `File`. It also declares two field signatures: `parent` and `contents`.

Every type signature represents a set of values, called *atoms*. The built-in integer type is an implicitly defined type signature, so the integer values are also considered to be atoms.

Field signatures represent relations. The two field signatures in the example above are binary relations, but relations of higher arity are also possible. A field declaration specifies the set of possible values for the field after the colon: the parent of any file system object must be a directory; while the contents of any directory can be arbitrary file system objects. Additionally, the field declaration specifies a *multiplicity* that restricts the number of atoms in the field's value. The default multiplicity for field declarations is `one`.

mult `:=`	
`set`	Any number of elements.
`some`	At least one element.
`lone`	At most one element.
`one`	Exactly one element.

Table 2.1.: Multiplicities

Inheritance of Alloy type signatures work similarly to Java: only single inheritance is allowed. Just like a Java object cannot be of multiple unrelated types; an Alloy atom cannot be a member of two unrelated type signatures. For example, intersection of the set of directories and the set of files is empty. The `abstract` modifier specifies that every atom in the set of file system objects is either a `Dir` or a `File`.

In addition to the signatures, an Alloy model contains a set of fact declaration. For example, the following fact expresses that the contents of a directory must specify that directory as their parent:

```
fact { all d: Dir, o: d.contents | o.parent = d }
```

An Alloy *instance* assigns concrete relations to the signatures. That is, for each type signature an instance gives the set of atoms; and for each field signature it gives a relation that matches the type and multiplicity of the field declaration. Fact declarations restrict the set of possible instances: it only contains those instances for which all the facts are valid formulas. If this set is empty, the model is inconsistent.

Finally, an Alloy model contains an instruction to either show all instances that satisfy a given predicate; or to show all instances that are counterexamples to a given assertion. Since the Alloy logic is not decidable in general, this analysis requires the user to specify an upper bound on the number of atoms for each type signature. Such a finite scope makes the analysis decidable, allowing fully automatic checks of the Alloy model.

Alloy has two kinds of expressions: relational expressions evaluate to a relation; and formulas evaluate to a Boolean truth value.

Our translation to JML will deal with individual formulas, not whole Alloy models. As such, we give a full list of expressions in the Alloy language. Only the syntax is given here. Where the semantics are unclear given an operator name, we will explain the semantics of the operator in chapter 5, where we also give the translation to JML.

expr `:=`	
identifier	variable or signature
number	integer literal
expr `+` *expr*	set union
expr `&` *expr*	set intersection
expr `-` *expr*	set difference
expr `.` *expr*	relational join
expr `[` *expr* `]`	relational join
expr `++` *expr*	relational override
expr [*mult*]`->`[*mult*] *expr*	Cartesian product
`~`*expr*	transpose
expr `<:` *expr*	domain restriction
expr `:>` *expr*	range restriction
`{`*varDeclList* `|` *formula*`}`	set comprehension
formula `implies` *expr* `else` *expr*	conditional
`#`*expr*	cardinality
`sum` *varDeclList* `|` *formula*	sum quantifier
`^`*expr*	transitive closure
`*`*expr*	reflexive and transitive closure

Table 2.2.: Alloy relational expression syntax

An Alloy variable declaration *varDecl* consists of a variable name, an optional multiplicity, and a relational expression specifying the quantifier range:

varDecl `:=` *identifier* `:` [*mult*] *expr*
varDeclList `:=` *varDecl* [`,` *varDeclList*]

formula `:=`	
expr [`not`] `in` *expr*	subset
expr [`not`] `=` *expr*	set equality
expr [`not`] *compOp* *expr*	integer comparison
`some` *expr*	non-empty
`one` *expr*	singleton
`no` *expr*	empty
`lone` *expr*	empty or singleton
`not` *expr*	negation
formula `and` *formula*	conjunction
formula `or` *formula*	disjunction
formula `iff` *formula*	logical equivalence
formula `implies` *formula*	implication
formula `implies` *formula* `else` *formula*	conditional
`all` *varDeclList* `|` *formula*	universal quantifier
`some` *varDeclList* `|` *formula*	existential quantifier
mult *varDeclList* `|` *formula*	quantifier with multiplicity

Table 2.3.: Alloy formulas

2.2. JML

In this section, we will give an overview of the Java Modeling Language [LPC+11]. We will only look at the subset of JML that is used by our translation of Alloy.

2.2.1. JML expression syntax

The JML expression syntax is an extension of the Java expression syntax. In addition to the standard Java logical operators `!`, `||` and `&&`, JML adds the implication operator `==>` and the equivalence operator `<==>`. The implication `a ==> b` is semantically equivalent to `!a || b`; it is a short-circuiting operator. The equivalence operator `a <==> b` works the same as the equality operator `a == b` on Boolean values, except that it has a lower operator precedence.

The JML expression syntax also adds several new keywords. To avoid conflicts with the names of Java program variables, these JML keywords start with a backslash. The following JML constructs are of importance to our translation:

`\fresh(`x`)`	Returns whether the object x is newly created in the post-state
`\old(`x`)`	Evaluates the expression x in the pre-state
`\invariant_for(`x`)`	Returns whether the invariant for object x is satisfied
`\result`	Refers to the return value of the method
`(\exists T v;` p`)`	Existential quantifier
`(\forall T v;` p`)`	Universal quantifier
`(\num_of T v;` p`)`	Number of objects `v` for which the expression p evaluates to `true`
`(\sum T v;` p`;` e`)`	Sum of the e for those objects `v` where p evaluates to `true`

Table 2.4.: JML constructs

Unlike Java, JML does not have exception handling. However, JML expressions can still throw an exception instead of evaluating to a value. An example is a member access on a null reference, which causes a `NullPointerException`. To handle these cases, JML uses strict validity semantics [Cha07]. Under these semantics, an exception such as the

NullPointerException causes the whole assertion will be interpreted as invalid. When discussing JML semantics, we will say that an expression is *well-defined* if the evaluation does not cause an exception.

2.2.2. JML contracts

JML contracts are used to specify the behavior of Java methods.

Here is an example contract for a method that calculates the square root of an integer:

```
//@ public normal_behavior
//@ requires i >= 0;
//@ ensures \result * \result <= i;
//@ ensures (\result + 1) * (\result + 1) > i;
public static /*@ pure @*/ int sqrt(int i) {
  ...
}
```

The requires clause specifies a precondition: the contract of this method only applies for non-negative inputs. For negative inputs, the behavior of our method is left unspecified. Methods may have multiple specification cases; an additional case could be used to specify that the method must throw an exception on invalid inputs.

The ensures clause specifies a postcondition. If multiple clauses exist, the postcondition of the method is the conjunction of all the ensures clauses.

The modifies clause specifies a frame condition: it lists the storage locations that may be modified by the method. A method marked as pure must not modify the heap at all.

The accessible clause can be used to specify which storage locations are read by the method.

The measured_by clause is used for recursive methods: it specifies an expression that computes an non-negative integer value. This value must decrease in recursive invocations of the method. This is used to prove termination of the recursive calls.

Finally, the invariant clause can be used anywhere within a class (not in method contracts). It specifies an object invariant that is established by the object's methods (unless the method is marked with the helper modifier); and also is required as a precondition.

2.2.3. Query Methods

A method marked as pure may be used in specification clauses. For example:

```
//@ requires sqrt(i) < n;
```

Model methods are methods that exist in the specification only, and do not have any implementation. They usually are pure and can be used as an abstraction of the program state within specifications.

2.2.4. JML*

KeY uses a modified version of JML, which is called JML*. It supports some additional constructs that are not present in JML, for example, it treats location sets (JML data groups) as values and provides operators to combine such sets [Wei11, 3.3]. KeY also does not implement all JML features.

Relevant to this work is a semantic difference in the interpretation of quantifiers between JML and JML*: The range of JML quantifiers extends over all objects of the given variable type, including objects that are not yet created [LPC$^+$11, 12.4.24.6]. In JML*, the quantifier range includes only those objects which are created in the current heap state.

For example, consider the following invariant that asserts that all doubly-linked list nodes that have the current node as a successor must be the predecessor of the current node:

```
invariant (\forall ListNode n; n.next == this ==> this.prev == n);
```

In JML, the quantifier range includes future `ListNode` instances that are not yet created in the current heap state. For these, the semantics of the expression `n.next` is unclear in the JML reference manual. In the Daniel Bruns' formalization of JML [Bru09], the expression `n.next` is not well-defined, and so is the the whole quantifier. Under strict validity, this means the whole invariant is invalid.

In JML*, the example expression is always well-defined as all values in the JML* quantifier range are non-null and created in the current heap state.

Our translation will assume the JML* semantics for the range of quantifiers. Apart from this semantic difference, the output of our translation function will be standard JML.

To obtain the JML* quantifier semantics in JML, we could write quantifiers using a range predicate:

```
\forall Object obj; \created(obj); ...
```

`\created` is a proposed JML construct that returns whether the object has been created in the current heap state. It is currently not part of the JML standard.

3. Alloy Specification Input Format

JML uses annotations in Java code files. When translating from Alloy to JML, we need a mapping from Alloy signatures to Java classes. Our tool Alloy2JML will derive this mapping implicitly from Java source code.

To embed Alloy specifications in Java code, we follow a similar approach to JML and introduce a new type of comment. Comments of the form `//$` and `/*$ */` are expected to contain specifications written in Alloy. The possible Alloy specification clauses are listed in table 3.1.

We also allow JML specifications using the usual JML syntax with `//@` and `/*@ @*/` comments. JML comments are mostly ignored by the translation and are copied as-is to the output file. This allows mixing of both specification languages within the same Java class. However, a few JML modifiers such as `nullable` and `non_null` have an effect on the translation of Alloy specifications.

`//$ requires`	Method precondition
`//$ ensures`	Method postcondition
`//$ modifies`	Method frame condition
`//$ invariant`	Instance invariant
`//$ pred`	Alloy predicate

Table 3.1.: Alloy contract clauses

An alternative syntax for Alloy specifications would have been to use Java annotations, for example `@Requires("element in head.*next"))`. This approach is taken by JForge [DCJ06]. However, this syntax gets cumbersome for more complicated specifications: Java does not allow duplicate annotations [Dar11]. This means that multiple preconditions would have to be specified within a single annotation by passing an array of strings. Moreover, Alloy expressions involving quantifiers often are written on multiple lines for improved readability, which would require string concatenation syntax as Java lacks multi-line string literals.

3.1. Alloy signatures for Java classes

To enable Alloy expressions to refer to Java fields, we define a mapping from Java class definitions to Alloy signatures. This allows us to interpret the semantics of the Alloy expressions using these generated signatures.

Java type	**Alloy signature**
`C` (class-type)	`lone C` (`C` is declared as type signature)
`int`	`Int`
`boolean`	`util/boolean`
`T[]`	`Array`
`I` (interface-type)	`lone I` (`I` is declared as subset signature that is equal to the union of all classes implementing the interface)
`null` literal	`none`

Table 3.2.: Correspondence between Java types and Alloy signatures

As a first step, every Java class is converted to an Alloy signature:

$\texttt{class C extends B \{\}} \quad \hookrightarrow \quad \texttt{sig C extends B \{\}}$

The signature `abstract sig Object {}` is used to represent the `java.lang.Object` base class.

Every Java field within such a class is converted to a pair of Alloy fields:

```
                member : lone T,
T member;  ↪
                member' : lone T,
```

Within the postcondition of a method (`ensures` clause), the Alloy field `member` represents the value of the Java field in the pre-state; and `member'` refers the value in the post-state. In all other contexts (preconditions, invariants, and predicates), there are no separate pre- and post-states available. Only the `member` field is available in these contexts, it is an error to refer to `member'`.

The fields are declared as `lone` to allow an empty set as a value. Empty sets in Alloy are used to represents null references in Java. This is the same convention as used by TestEra [KYZ$^+$11]. It differs from JForge [DCJ06], which uses a `null` atom to represent the null reference.

To express that a certain member must not be `null` when calling a method, the Alloy precondition would look like this:
`requires some this.member`
Usage of the `null` keyword within an Alloy specification is invalid.

The translation of Java types to Alloy types is given in table 3.2. Note that floating-point types and integers other than `int` are currently unsupported by Alloy2JML. A field of type `int` would be translated as follows:

```
                  member : lone Int,
int member;  ↪
                  member' : Int,
```

In the post-state, the member is not marked as `lone` because Java integers are not nullable. However, we always mark the pre-state as `lone` no matter which Java type the field uses. This is because the Alloy model includes objects that get created during the execution of the method, and thus exist only in the post-state in the JML world. In the Alloy model, these fresh objects have all of their pre-state members set to the empty set.

For arrays, we use the following type signature:

```
sig Array extends Object {
  length : Int,
  elements : Int -> lone univ,
  elements' : Int -> lone univ
}
```

The `elements` relation maps from array indices to the values stored in the array. For an out-of-bounds index, the `elements` relation returns the empty set.

As Java allows array elements to be `null`, it is possible that valid indices also map to the empty set. This differs from Alloy's built-in sequence type "`seq`".

3.2. Object creation

In this section, we will explain how object creation affects the Alloy model. The previous section already touched on this topic: pre-state members always are `lone` because the Alloy model include objects that are created by the method.

We now define more explicitly: in the Alloy model, the type signature `C` contains all object instances of `class C`, or its subclasses, in the current heap state. For postconditions, the current heap state is the post-state, so the model includes objects that do not yet exist in the pre-state. However, objects from other heap states are not contained in the model – all instances in a type signature `C` exist in the post-state. Within preconditions, the current heap state is the pre-state, so all instances in the type signature exists in the pre-state.

This definition is important when quantifying over a type signature, for example "`all obj : C | F`" – we want this formula to refer to all objects created by the program so far, not to the infinite set of possible objects. The definition was chosen this way because it directly corresponds to the way JML* quantifiers work in KeY. In fact, if our definition for the contents of a type signature contained any additional objects, we would be unable to use JML* quantifiers in the translation as they could not make any statements about the additional objects.

Within postconditions, it is often necessary to specify that certain objects are freshly allocated. In JML, this is accomplished using the "`\fresh(obj)`" expression. To make this feature available in postconditions specified using Alloy, we define the set signature `fresh` to represent the set of newly allocated objects.

```
sig fresh in Object {}
```

This signature is defined to contain the set of objects that are not yet created in the pre-state, but are created in the post-state. The use of the `fresh` signature is only valid within postconditions.

3.3. Contract Clauses

Semantic analysis of Alloy formulas in contract clauses occurs as if the formula was enclosed in a predicate of the form:
`pred C.predicateName[...] { ...}`
where `C` is the name of the current class.

In particular, the "implicit this" Alloy feature is not available in this context. To access an instance variable, it must be explicitly qualified with a this reference (`this.member`). The identifier `member` alone refers to the binary relation from the current class `C` to the type of the field.

For example, the Alloy formula
`//$ requires no this.member`
translates to
`//@ requires this.member == null;`

On the other hand, the Alloy formula

```
//$ requires no member
```

translates to

```
//@ requires (\forall C c; c.member == null);
```

3.3.1. Preconditions

Preconditions are specified using "`//$ requires` *alloy-formula*", and directly correspond to JML preconditions. They must be declared within a class body, in front of the method they apply to.

Within the formula of a precondition, the method parameters are available as Alloy variables (as if they were parameters of the enclosing predicate). As with fields, a null reference in Java corresponds to an empty set in Alloy.

As in JML, the precondition formula acts on the pre-state heap. If any Alloy type signatures are used in the formula, they do not refer to the whole set of objects (of that type), but only to those that already exist in the pre-state. This means that type signature `C` used in a precondition is equivalent to `C - fresh` in a postcondition.

It is an error to explicit refer to the `fresh` signature within a precondition; and also an error to refer to any post-state fields (`member'`).

Together, these rules prevent preconditions from accessing the post-state; the preconditions can only depend on the pre-state.

3.3.2. Postconditions

Postconditions are specified using "`//$ ensures` *alloy-formula*", and directly correspond to JML postconditions.

They have access to method parameters, and additionally can access a special Alloy variable `result` that holds the method's return value.

Postconditions can refer to both the pre-state and the post-state. Type signatures used within a postcondition refer to the post-state. Field names suffixed with the prime character (`'`) refer to post-state; the plain field names refer to the pre-state.

When using predicates within a postcondition, there are two options: If the predicate name is followed by a prime character, it is evaluated in the post-state. This means that any references to fields within the predicate are accessing the field in the post-state, even though the predicate definition itself does not use the prime character. In this case, type signatures within the predicate refer to objects in the post-state.

The other option is to use the predicate name without a prime character. This causes evaluation of the predicate in the pre-state: any fields refer to the pre-state values, and type signatures refer to the objects in the pre-state, as if the expression was written within a precondition.

This syntax of distinguishing pre- and post-state is more flexible than the JML `\old(...)` expression. In particular, it is possible to a pre-state field of an object stored in a post-state variable: `this.next'.data`

This cannot be directly expressed in JML as the `\old` keyword cannot be applied to the `data` field alone, only to the full expression. However, a formula involving such a construct can be translated by introducing a quantifier:

`no this.next'.data`
translates to
`(\forall Node obj; this.next == obj && !\fresh(obj) ==> \old(obj.data) == null)`

In this case, the `!\fresh(obj)` safety check is necessary to avoid accessing `\old(obj.data)` on a newly created object. In the Alloy model, such an access is well-defined to produce an empty set. But in JML, such an access is undefined and would make the whole formula undefined, similar to accessing a member on a null reference.

3.3.3. Modifies clause

The modifies clause specifies the list of memory locations that may get modified by the method. As with pre- and postconditions, the modifies clause must be specified in front of a method declaration.

Syntax: `//$ modifies` *location-list*
location-list ::= *location* | *location* `,` *location-list*
location ::= *alloy-expression* `.` *identifier*

Each location is specified using an Alloy relational expression specifying a set of object instances, and an identifier specifying the field name.

Semantic analysis of the *alloy-expression* occurs as within preconditions. In particular, method parameters may be used; and any access to the post-state is illegal.

As an example, a method that adds/removes nodes from a binary tree can be specified like this:

```
//@ modifies this.*(left+right).left, this.*(left+right).right
```

A similar effect could be achieved using postconditions:

```
//@ ensures all t : (Tree - fresh - this.*(left+right)) | t.left' = t.left
//@ ensures all t : (Tree - fresh - this.*(left+right)) | t.right' = t.right
```

The example modifies clause is equivalent to these postconditions, except that the modifies clause also implies that no other fields in the program are changed.

3.3.4. Invariants

Invariants defined using "`//$ invariant` *alloy-formula*" correspond to JML instance invariants. They must be declared within in a class body.

Invariants are implicitly part of method contracts unless the method is declared using the JML `helper` modifier.

It is an error to refer to any post-state fields or post-state predicates from within an invariant. If type signatures are used in an invariant, their meaning depends on the current heap state at the place where the invariant is used.

The conjunction of all invariants in a class can be referred to using the Alloy formula `invariantFor[obj]`, as if `invariantFor` was a predicate. This corresponds to the JML `\invariantFor(...)` construct.

3.3.5. Predicates

Predicates defined using `//$ pred` provide an abbreviation for an Alloy formula. They can be referred to by all Alloy contracts in the input file.

Each predicate is translated into a model method of the same name; and any usage of the predicate will be translated to an invocation of the model method. When mixing both JML and Alloy specifications within a single file, it is possible to invoke Alloy predicates within a JML specification.

Example:

```
/*$ pred Tree.isSorted {
      all d : this.*(left+right) {
        all vl : d.left.*(left+right).value  | vl < d.value
        all vr : d.right.*(left+right).value | vr > d.value
      }
    }
*/
```

3.4. Restrictions on the allowable Alloy expressions

Not all Alloy expressions are supported by our tool. In this section, we list the restrictions on the accepted Alloy expressions. Some of these restrictions are fundamental to the approach used by our translation. In other cases, Alloy language features were merely not implemented due to time constraints.

The first fundamental restriction is that we do not support sets that contain both objects and primitive integers or booleans. This restriction exists due to the Java type system: When the translation to JML introduces a quantifier for an Alloy set, the translation needs to pick a type for the bound variable that encompasses all values that are possibly contained in the Alloy set. This is impossible for sets mixing objects and primitive values, as no suitable Java type exists.

Theoretically, this issue could be fixed by introducing multiple quantifiers instead: one for objects, and one for each primitive type. However, we have not done so because this can quickly lead to an exponential increase in the size of the generated JML when such quantifiers are nested, and because sets with both objects and primitive values are rarely useful when writing specifications for Java programs.

As a consequence of this first restriction, the built-in `univ` relation is unsupported. Moreover, the built-in `iden` relation has slightly different semantics: it is interpreted as the relation `{a : Object, b : Object | a = b}`. In comparison to standard Alloy, this means `iden` is missing integers and booleans.

The second fundamental restriction is that higher-order quantification is not supported. Higher-order quantification occurs in Alloy when a variable declaration in an Alloy quantifier uses a multiplicity other than `one`. The Alloy Analyzer supports higher-order quantification only in cases where skolemization can be used to eliminate the quantifier. Alloy2JML has no support for higher-order quantification at all.

Additionally, we do not support quantification with a range expression having arity greater than 1; for example "`all n : iden | n = ~n`". Such quantifiers always involve higher-order quantification in Alloy[1], so they are unsupported in Alloy2JML.

Another fundamental restriction applies to the arguments of predicates: because predicates are converted to Java methods, we cannot pass any relations/sets to those methods. For this reason, the argument expressions passed to an Alloy predicate must evaluate to the empty set or a singleton set. In the Alloy2JML implementation, predicate arguments must be simple expressions using the following syntax constructs:

simple-expr :=	
identifier	bound variable or program variable
number	integer literal
simple-expr . *identifer*	relational join of simple expression with a field

All simple expressions are statically known to evaluate to an empty or singleton set.

[1]The default multiplicity for quantification with arity>1 is `set`, and the multiplicity `one` is unavailable.

In principle, Alloy functions are similar to Alloy predicates. However, we were unable to add support for functions due to time constraints. Our planned approach to implementing functions is simple: we treat functions as predicates that accept a tuple. The predicate would be valid iff that tuple is a member of the relation returned by the function. This way, the same restrictions apply to the function arguments as for predicate arguments, but the return value can be an arbitrary Alloy relation.

Regarding the transitive closure operator, there is a small restriction: if the relation being closed over accesses the heap, it must refer to at most one of the heap states (pre-state or post-state), not both. As such, the transitive closure "`*(left' + right)`" (where `left` and `right` are fields) cannot be translated to JML.

Of the built-in Alloy functions, we only support a couple functions like `add[]` and `sub[]`. Most of these would be simple to translate into the directly equivalent JML/Java constructs; however we have not done so due to time constraints.

Finally, the Alloy keywords "`let`" and "`disj`" are unsupported. The use of any unsupported construct will result in the translation aborting with an error message.

4. Output Alloy model

Apart from the translation to JML, our tool Alloy2JML will also generate an Alloy model from the Java code and its Alloy annotations. This model can be used with the Alloy Analyzer to visualize the program specification. This way, mistakes in the specification can be detected before attempting to verify the program against it.

The Alloy model is constructed by creating Alloy signatures for the Java classes as described in section 3.1. Additionally, our tool generates some fact declarations to account for the the semantics of Java object creation. For every Java field `T member` within every class `C`, we generate the following two facts:

```
all obj : C & fresh | no obj.member
all obj : C | no (obj.member & fresh)
```

The former fact enforces that objects that are not yet created in the pre-state, do not have any field values in the pre-state. The latter fact models that fields in the pre-state cannot refer to objects that get created later.

Alloy predicates from the input Java file are copied into the Alloy model as-is. Additionally, a post-state version of these predicates is generated by substituting all references to pre-state fields with the corresponding post-state fields.

Invariants are handled in the same fashion: for every class `C`, a predicate with the name "`invariantFor`*C*" is constructed by creating the conjunction of all invariant clauses in that class.

Finally, for every method that has a specification, a pair of predicates "`method_pre`" and "`method_post`" are created. Both predicates have a parameter list corresponding to the method's parameters. The `pre` predicate encodes the method's preconditions. Apart from a conjunction of the user-specified `requires` clauses, it also requires that any object parameters exist in the pre-state heap, as well as the object invariant. The `post` predicates encodes the method's postcondition. Any `modifies` clauses are converted into equivalent postconditions and included in the `post` predicate.

As an example, we will use an unbalanced binary search tree:

```
final class Tree {
  /*@ nullable @*/ Tree left, right;

  int value;
  /*$ invariant {
        all d : this.*(left+right) {
            all dl : d.left.*(left+right)  | dl.value < d.value
            all dr : d.right.*(left+right) | dr.value > d.value
          }
      }
  */

  //$ ensures this.*(left' + right').value' = this.*(left + right).value + v
  //$ modifies this.*(left+right).left, this.*(left+right).right
  public void add(int v) { /* ... */ }
}
```

When passed through Alloy2JML, the Java code is converted to the following Alloy model:

```
abstract sig Object {}
sig fresh in Object {}
sig Tree extends Object {
  left  : lone Tree,  left' : lone Tree,
  right : lone Tree,  right' : lone Tree,
  value : lone Int,   value' : Int,
}
fact {
  // Objects that are not created in pre-state do not have any field values
  all obj : Tree & fresh { no obj.left and no obj.right and no obj.value }
  // Fields in the pre-state cannot refer to objects that get created later
  all obj : Tree { no (obj.left & fresh) and no (obj.right & fresh) }
}

pred Object.invariantForObject { }
pred Object.invariantForObject' { }
pred Tree.invariantForTree {
  invariantForObject[this]
  all d : this.*(left+right) {
    all dl : d.left.*(left+right)  | dl.value < d.value
    all dr : d.right.*(left+right) | dr.value > d.value
  }
}
pred Tree.invariantForTree' {
  invariantForObject'[this]
  all d : this.*(left'+right') {
    all dl : d.left'.*(left'+right')  | dl.value' < d.value'
    all dr : d.right'.*(left'+right') | dr.value' > d.value'
  }
}
...
```

To visualize the invariant, the user can use the Alloy Analyzer to write an additional predicate that restricts the state space to be visualized:

```
one sig Root extends Tree {}
pred Show {
  view_invariant // generated predicate that disables the post-state signatures
      by requiring them to be equal to the pre-state signature
  Tree = Root.*(left + right) // only view a single tree
    invariantForTree[Root]
    Tree.value = 1 + 2 // values in tree = {1,2}
}
run Show for 3
```

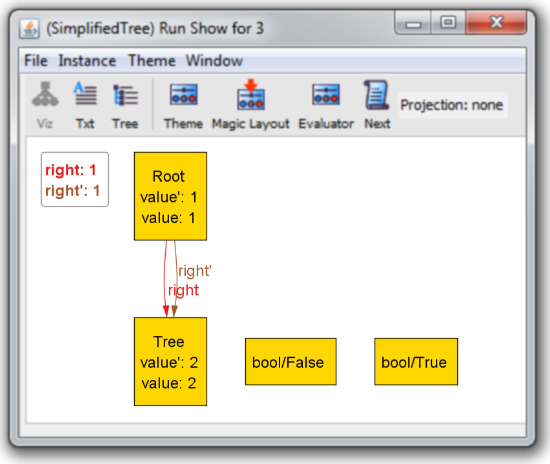

Figure 4.1.: Visualization of a Tree instance

Using the Alloy Analyzer to view the satisfying instances, we see two possible instances: one where the value 1 is the root node and 2 its right child; and one where the value 2 is the root node and 1 the left child. Figure 4.1 shows the former case.

If we introduce a mistake in the invariant, the Alloy Analyzer may find additional instances that were not intended to be valid binary search trees. For example, consider a modification to the tree that is intended to allow duplicate values (implementing a multi-set). If this is done by changing the value comparisons in the invariant to also allow equal values, we would not only gain the instances with duplicate values as intended, but also instances that are no trees at all, as the modified invariant no longer prevents cycles from forming.

As for method contracts, the contract of the `Tree.add()` method is converted to the following pair of predicates:

```
pred Tree.add_pre[v : Int] {
  this not in fresh
  invariantForTree[this]
}
pred Tree.add_post[v : Int] {
  no result // void
  invariantForTree'[this]
  this.*(left' + right').value' = this.*(left + right).value + v
  // The following formulas are created from
  // 'modifies this.*(left+right).left, this.*(left+right).right':
  all o : Tree - fresh - this.*(left+right) | o.left' = o.left
  all o : Tree - fresh - this.*(left+right) | o.right' = o.right
  all o : Tree - fresh | o.value' = o.value
}
```

Here, we can use the Alloy Analyzer to visualize the possible invocations of the add method. To restrict the set of instances to a manageable size, we restrict the input to the method in a similar way as we did for the invariant:

```
pred A {
  // All tree nodes in the model should be involved in the tree being modified:
  Tree = Root.*(left + right + left' + right')
  Root.*(left+right).value = 1 + 2  // initial values in tree = {1,2}
  Root.value = 2 // pick the initial tree where 2 is the root node
  Root.add_pre[3] // consider only trees that satisfy the precondition
  Root.add_post[3] // restrict the post-state to satisfy the postcondition
}
run A for 4
```

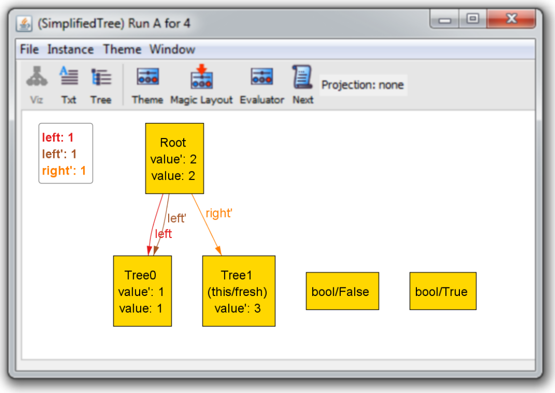

Figure 4.2.: Visualization of the expected result of Tree.add() call

Figure 4.2 shows one of the instances found by the Alloy Analyzer. However, there are many other instances where existing tree nodes are being replaced with new instances. This means our specification was underspecified, a more strict specification could demand that we preserve the existing objects in the tree and only create a single new object. For the purpose of our Java specification, this kind of underspecification is fine. However, it makes visualization of possible results of the method difficult, as the result space allowed by the specification is large. We can add another clause to our predicate to narrow down the result space: `Root.*(left+right) in Root.*(left’+right’)`

Now there are two satisfying instances: one where the added node is `fresh`, and one where it isn’t. It turns out that this is a real bug in our specification: it allows the `add()` method to steal leaf nodes from unrelated tree instances instead of allocating new objects. If further `add()` operations add children to the stolen node, the values stored in the unrelated tree change.

Without using the Alloy Analyzer to explore the space of possible implementations for the specification, it can be difficult to spot unintentional underspecification of this kind. The problem may go unnoticed for a long time as long as programs use only a single tree instance. Only when a program involves multiple trees, verification of the program may become impossible using the problematic specification. With the Alloy Analyzer and Alloy2JML, the problem can be discovered early in the specification phase, before any implementations are written or verified.

5. Translation Function

The translation from Alloy to JML is defined using two functions $\mathcal{B}$ and $\mathcal{E}$ that are defined recursively on the Alloy syntax tree. The function $\mathcal{B}$ is used on Alloy formulas; whereas $\mathcal{E}$ is used for relational expressions. An additional third function $\mathcal{I}$ is used for integer expressions. It is defined in terms of the $\mathcal{E}$ function and is not recursive itself.

The translation using these two functions is followed by a simplification pass. This step dramatically increases the readability of the resulting JML formulas, as the translation introduces lots of quantifiers that get subsequently removed by the simplification rules. Readability is important as correctness proofs in the KeY theorem prover may fail and need manual intervention by the user. In such a case, the more readable translation result can help the user to relate her original specification to the state of the theorem prover.

5.1. Translation Context

The translation context captures various information about context the in which the to-be-translated Alloy expression was used:

- The translation context keeps track of whether the expression occurs within a post-condition. This affects the creation of the JML `\old(...)` expression.
- The translation context comes with a mapping from Alloy signatures to Java fields. This mapping is usually derived from the Java program as described in the previous chapter. By providing this mapping in a different manner, the translation function could be re-used in a different context.
- The context also maps Alloy predicates to Java model methods. This mapping is initialized with the translation of predicates to model methods as described in section 5.6.2.
- Finally, the translation context maintains a mapping from bound Alloy variables to JML variables. Initially, it only contains the program variables (`this` and method parameters) that are in scope for the specification clause. When the recursive translation function encounters Alloy quantifiers, it will extend this mapping with the additional variables.

In the implementation, the translation context is additionally used to store the state necessary for creating unique variable names.

5.2. Translation Predicates

A *translation predicate* is a function that, when given a JML object expression, returns a JML boolean expression. The boolean expression will evaluate to `true` when the condition encoded by the predicate is satisfied by the given object, and false otherwise.

A translation predicate is called *well-formed* if the following conditions hold:

- If the input object expression does not cause an exception to be raised, the output boolean expression does not cause an exception to be raised.
- If the input object expression evaluates to a null reference, the output boolean expression evaluates to `false`.
- The semantics of the predicate do not depend on the heap state, so that it does not matter whether it is evaluated within a JML `\old` expression.

In JML with strict validity semantics, any exception (such as `NullPointerException`) will cause the expression to be considered undefined and the whole assertion will be interpreted as invalid.

For this reason, the recursive translation function will ensure that its output expressions will never cause exceptions, and always be well-defined. Well-formed predicates play a crucial role in ensuring that this guarantee is upheld.

An example of a well-formed predicate is the *nonnull* predicate:

$$\textit{nonnull}(e) := \texttt{(}e\ \texttt{!= null)}$$

The *nonnull* predicate produces a JML expression that evaluates `true` for any object, while evaluating to `false` for null references as required for well-formed predicates.

The most commonly used predicate is *lift*(`obj`), which evaluates to `true` if the object passed into the predicate is the same as `obj`:

$$\textit{lift}(\texttt{obj})(e) := \texttt{(}e\ \texttt{== obj)}$$

To ensure that the predicate *lift*(`obj`) is well-formed, the expression `obj` must be statically known to be exception-free and non-null.

The *nonnull* predicate and the *lift*(...) predicates are sufficient for the translation of any user-specified contract clause. However, when generating the recursive contract for query methods created for the translation of the transitive closure (see chapter 6), a third kind of translation predicate will come into play.

5.3. Translation of relational expressions

The translation of Alloy relational expressions is performed using the function:
$\mathcal{E}«\texttt{r}\|p_1,\ldots,p_n»_c$

The $\mathcal{E}$ function takes 3 arguments, and produces a JML boolean expression as output.

- `r` is the Alloy relational expression to be translated.
- $p_1,\ldots,p_n$ is a list of translation predicates. The length of the list must be equal to the arity of the relation `r`, and the predicates must be well-formed.
- c is the translation context

The resulting JML expression will evaluate to `true` if and only if there exists a tuple t in the Alloy relation `r` so that every predicate is satisfied by the given item in the tuple t.

$$\mathcal{E}«\mathtt{r}\|p_1, \ldots, p_n»_c \text{ evaluates to } \mathtt{true} \iff \exists(t_1, \ldots, t_n) \in \mathtt{r}\colon\ p_1(t_1) \wedge \ldots \wedge p_n(t_n)$$

Otherwise, $\mathcal{E}«\mathtt{r}\|p_1, \ldots, p_n»_c$ evaluates to `false`. The output expression is guaranteed to never throw an exception.

For relations of arity 1, the usage of the *lift*(...) predicate with the $\mathcal{E}$ function corresponds to element-of semantics:

$$\begin{aligned}\mathcal{E}«\mathtt{r}\|\mathit{lift}(\mathtt{obj})»_c \text{ evaluates to } \mathtt{true} &\iff \exists e \in \mathtt{r}\colon\ \mathit{lift}(\mathtt{obj})(e) \text{ evaluates to } \mathtt{true} \\ &\iff \exists e \in \mathtt{r}\colon\ (e \texttt{ == } \mathtt{obj}) \text{ evaluates to } \mathtt{true} \\ &\iff \mathtt{obj} \in \mathtt{r}\end{aligned}$$

The usage of the *nonnull* predicate with the $\mathcal{E}$ function checks whether an Alloy set is non-empty:

$$\begin{aligned}\mathcal{E}«\mathtt{r}\|\mathit{nonnull}»_c \text{ evaluates to } \mathtt{true} &\iff \exists e \in \mathtt{r}\colon\ \mathit{nonnull}(e) \text{ evaluates to } \mathtt{true} \\ &\iff \exists e \in \mathtt{r}\colon\ (e \texttt{ != null}) \text{ evaluates to } \mathtt{true} \\ &\iff \mathtt{r} \neq \emptyset\end{aligned}$$

As e is an atom in an Alloy set, it represents a JML object and thus cannot be the null reference.

For Alloy relational expressions of higher arity, the list of predicates may contain both *lift*(...) and *nonnull* at the same time. This makes the existential translation function $\mathcal{E}$ more flexible than an approach using two separate translation functions for the element-of and the empty-set cases. We found that this additional flexibility helps produce JML expressions where the redundant quantifiers are more easily eliminated by the simplification step.

5.3.1. Type function

The translation makes use of the type function $\mathcal{T}_i[\mathtt{r}]$. This function takes an Alloy relational expression `r`, and returns the Java type for column number i in the relation. This uses the type mapping from table 3.2.

If the Alloy relation is a union type of different signatures, the function $\mathcal{T}$ will return the most specific common base type.

In the Alloy2JML implementation, we use the Alloy Analyzer's Java API to perform semantic analysis of the Alloy expression, so that we can determine the arity and column types of the Alloy relational expressions.

5.3.2. Variables

Alloy variables are translated to JML by applying the translation predicate to the corresponding JML variable:

$$\mathcal{E}«\mathtt{v}\|p_1»_c := p_1(\mathit{var}_c(\mathtt{v})))$$

The notation $\mathit{var}_c(\mathtt{v})$ represents the JML variable corresponding to the Alloy variable `v`.

There are two kinds of Alloy variables encountered by the translation: program variables (e.g. method parameters or the `this` pointer), or bound variables introduced by a quantifier. Program variables, viewed from Alloy, hold either the empty set (null reference) or a singleton set. Bound variables in Alloy always hold a singleton set, as our translation prohibits higher-order quantification (see chapter 3.4 for details).

Both program variables and bound variables are translated the same way, by applying the translation predicate. Because the predicate is well-formed, it will produce `false` when given a program variable holding a null reference. This is the expected result for an empty set, as the $\mathcal{E}$ function has existential semantics.

5.3.3. Type signatures

As described in chapter 3.1, Alloy type signatures directly correspond to Java classes. To translate the usage of a type signature to JML, we materialize the existential quantifier that is implicit in the definition of $\mathcal{E}$:

$$\mathcal{E}«\texttt{T}\|p_1»_c := \texttt{(\textbackslash exists T obj; } p_1\texttt{(obj))}$$

The name `obj` is a placeholder for a unique name generated by the translator. This is important to avoid name collisions, as the identifier `obj` may occur with a different meaning within the predicate.

JML* `\exists` only quantifies over the objects that exist in the current heap state, so this translation naturally matches the semantics for type signatures described in chapter 3.3.1.

The introduction of existential quantifiers, followed by an application of the corresponding translation predicate, is used in many translation rules. Within the quantifier body, we can operate on a single element as if we had used element-of semantics instead of existential semantics for the translation function. Note that the quantifiers introduced this way are often redundant: if the predicate p_1 is a predicate of the form *lift*(e), the quantifier body can only be true if the bound variable is equal to the expression e. This means that the simplification step following the translation will be able to eliminate the quantifier, resulting in the translation:

$$\mathcal{E}«\texttt{T}\|\mathit{lift}(e)»_c = \texttt{(}e\texttt{ instanceof T)}$$

This also applies to the translations of most other Alloy constructs: the vast majority of quantifiers introduced by the translation will be eliminated by the simplification step.

5.3.4. Field signatures

The usage of a field signature in Alloy results in a binary relation that maps objects to the field value.

$$\mathcal{E}«\texttt{member}\|p_1, p_2»_c := \texttt{(\textbackslash exists } \mathcal{T}_1[\texttt{member}] \texttt{ obj; } p_1\texttt{(obj) \&\& } p_2\texttt{(obj.member))}$$

$\mathcal{T}_1$[`member`] is the type of the first column of the binary relation, which is the class declaring the field `member`.

The generated expression is guaranteed to be well-defined: no NullPointerException can occur during the field access, as the existential quantifier ensures that `obj` is non-null and has the correct type.

If the value stored in `obj.member` is null, the Alloy relation corresponding to the member will not contain a tuple for that `obj`. The translated expression thus must return false.

This is the reason why we required well-formed predicates to evaluate to `false` when given a null reference: it allows us to directly pass the field to the predicate without generating an explicit null check. Explicit null checks inserted at this point would often be redundant, as the *lift*() predicate contains an implicit null check due to the comparison with a non-null field. By shifting the responsibility for these safety checks into the predicates, we delay the choice of whether to generate a check until more information is available to determine if the check is redundant.

If the translation context c indicates that we are currently translating a postcondition, and the member access does not use the prime suffix for accessing the post-state, the translation uses the JML `\old` operator:

$$\mathcal{E}«\texttt{member}\|p_1, p_2»_c := \texttt{\textbackslash old((\textbackslash exists } \mathcal{T}_1[\texttt{member}] \texttt{ obj; } p_1(\texttt{obj}) \texttt{ \&\& } p_2(\texttt{obj.member})\texttt{))}$$

It is critical that the `\old` operator is used outside the existential quantifier: it changes the semantics of the JML* quantifier so that it only quantifies over the objects that exist in the old heap state. If we were to use `\old` only around the `obj.member` access, we would end up accessing a field value of an object that does not exist yet. This would cause the body of the quantifier to evaluate to an undefined value.

Because the code generated by the predicates is within the scope of the `\old` expression as well, we require all well-formed predicates to have the same semantics no matter whether they run within `\old` or not. An alternative solution that does not require this restriction on predicates would be to use an explicit `\fresh(...)` check:

$$\texttt{(\textbackslash exists } \mathcal{T}_1[\texttt{member}] \texttt{ obj; } p_1(\texttt{obj}) \texttt{ \&\& !\textbackslash fresh(obj) \&\& } p_2(\texttt{\textbackslash old(obj.member)})\texttt{)}$$

5.3.5. Built-in relations and literals

As described in section 3.4, the built-in `iden` relation is restricted to objects and does not work for primitive types. For the same reason, the `univ` relation is not supported at all, as the set of all objects can already be referred to using the type signature `Object`.

$$\mathcal{E}«\texttt{none}\|p_1»_c := \texttt{false}$$
$$\mathcal{E}«\texttt{iden}\|p_1, p_2»_c := \texttt{(\textbackslash exists Object obj; } p_1(\texttt{obj}) \texttt{ \&\& } p_2(\texttt{obj})\texttt{)}$$

For the built-in relations `bool/True` and `bool/False`, as well as for integer literals, the translation trivially applies the translation predicate to the literal value:

$$\mathcal{E}«\texttt{True}\|p_1»_c := p_1(\texttt{true})$$
$$\mathcal{E}«\texttt{False}\|p_1»_c := p_1(\texttt{false})$$
$$\mathcal{E}«num\|p_1»_c := p_1(num)$$

5.3.6. Set Union

As the $\mathcal{E}$ function has existential semantics, set union can be simply translated into a logical or:

$$\mathcal{E}«\texttt{a + b}\|p_1, \ldots, p_n»_c := (\mathcal{E}«\texttt{a}\|p_1, \ldots, p_n»_c \texttt{ || } \mathcal{E}«\texttt{b}\|p_1, \ldots, p_n»_c)$$

5.3.7. Set Intersection

The existential semantics of $\mathcal{E}$ don't play well with set intersection. We need to materialize the quantifier for all columns in the relation:

$$\begin{aligned}\mathcal{E}\langle\!\langle \texttt{a \& b} \| p_1,\dots,p_n \rangle\!\rangle_c := {} & \texttt{(\textbackslash exists } \mathcal{T}_1[\texttt{a\&b}]\ \texttt{o}_1\texttt{, ..., } \mathcal{T}_n[\texttt{a\&b}]\ \texttt{o}_n\texttt{;} \\ & p_1\texttt{(o}_1\texttt{) \&\& ... \&\& } p_n\texttt{(o}_n\texttt{) \&\&} \\ & \mathcal{E}\langle\!\langle \texttt{a} \| \mathit{lift}(\texttt{o}_1),\dots,\mathit{lift}(\texttt{o}_n) \rangle\!\rangle_c \texttt{ \&\& } \mathcal{E}\langle\!\langle \texttt{b} \| \mathit{lift}(\texttt{o}_1),\dots,\mathit{lift}(\texttt{o}_n) \rangle\!\rangle_c\texttt{)}\end{aligned}$$

We generate the existential quantifier for the whole tuple, and lift the newly introduced bound variables for use as translation predicates in the recursive call on $\mathcal{E}$. The variables introduced by the JML quantifiers are statically known to be non-null, so the $\mathit{lift}(\texttt{o}_i)$ predicates are well-formed.

To avoid name collisions, the $\texttt{o}_i$ are newly created unique identifiers.

5.3.8. Set Difference

Set difference works like set intersection, except that the boolean expression generated for `b` is negated:

$$\begin{aligned}\mathcal{E}\langle\!\langle \texttt{a - b} \| p_1,\dots,p_n \rangle\!\rangle_c := {} & \texttt{(\textbackslash exists } \mathcal{T}_1[\texttt{a-b}]\ \texttt{o}_1\texttt{, ..., } \mathcal{T}_n[\texttt{a-b}]\ \texttt{o}_n\texttt{;} \\ & p_1\texttt{(o}_1\texttt{) \&\& ... \&\& } p_n\texttt{(o}_n\texttt{) \&\&} \\ & \mathcal{E}\langle\!\langle \texttt{a} \| \mathit{lift}(\texttt{o}_1),\dots,\mathit{lift}(\texttt{o}_n) \rangle\!\rangle_c \texttt{ \&\& !}\mathcal{E}\langle\!\langle \texttt{b} \| \mathit{lift}(\texttt{o}_1),\dots,\mathit{lift}(\texttt{o}_n) \rangle\!\rangle_c\texttt{)}\end{aligned}$$

5.3.9. Relational Join

The relational join operator "." can join any two relations by matching the last column of the first relation with the first column of the second relation. That is, the relational join `a.b` contains a tuple $(x_1,\dots,x_n,y_1,\dots,y_m)$ if there exists an atom obj so that $(x_1,\dots,x_n,obj) \in \texttt{a}$ and $(obj,y_1,\dots,y_m) \in \texttt{b}$.

This corresponds to a simple Java member access if the first relation is a singleton set and the second relation is a field signature.

The translation works by introducing the existential quantifier for obj, and otherwise splitting the predicate list:

$$\begin{aligned}\mathcal{E}\langle\!\langle \texttt{a.b} \| p \rangle\!\rangle_c := {} & \texttt{(\textbackslash exists } \mathcal{T}_1[\texttt{b}]\ \texttt{obj; } \mathcal{E}\langle\!\langle \texttt{a} \| p_1,\dots,p_n,\mathit{lift}(\texttt{obj}) \rangle\!\rangle_c \\ & \qquad \texttt{\&\& } \mathcal{E}\langle\!\langle \texttt{b} \| \mathit{lift}(\texttt{obj}),p_{n+1},\dots,p_{n+m} \rangle\!\rangle_c\texttt{)} \\ & \text{where } \ n = \mathit{arity}(\texttt{a}) - 1 \ \text{ and } \ m = \mathit{arity}(\texttt{b}) - 1\end{aligned}$$

To avoid name collisions, `obj` is a newly created unique identifier.

5.3.10. Relational Override

The relational override of `a` by `b` contains all tuples in `b`, and additionally, any tuples of `a` whose first element is not the first element of a tuple in `b`.

It is useful for describing updates being applied to a field signature.

$$\begin{aligned}\mathcal{E}\langle\!\langle \texttt{a ++ b} \| p_1,\dots,p_n \rangle\!\rangle_c := {} & \texttt{(\textbackslash exists } \mathcal{T}_1[\texttt{a++b}]\ \texttt{t; } p_1(\texttt{t}) \texttt{ \&\& (}\mathcal{E}\langle\!\langle \texttt{b} \| \mathit{lift}(\texttt{t}),p_2,\dots,p_n \rangle\!\rangle_c \texttt{ ||} \\ & \texttt{(}\mathcal{E}\langle\!\langle \texttt{a} \| \mathit{lift}(\texttt{t}),p_2,\dots,p_n \rangle\!\rangle_c \texttt{ \&\& !}\mathcal{E}\langle\!\langle \texttt{b} \| \mathit{lift}(\texttt{t}),\underbrace{\mathit{nonnull},\dots}_{n-1 \text{ times}} \rangle\!\rangle_c\texttt{)))}\end{aligned}$$

The translation materializes the existential quantifier for the first column. To check for the existence of any tuple with the first element `t`, the *nonnull* predicate is used.

5.3.11. Cartesian Product

To determine whether the product of `a` and `b` contains a tuple satisfying the predicate list, we split the predicate list and check that each input relation individually:

$$\mathcal{E}«\texttt{a -> b}\|p_1,\dots,p_{n+m}»_c := (\mathcal{E}«\texttt{a}\|p_1,\dots,p_n»_c \;\texttt{\&\&}\; \mathcal{E}«\texttt{b}\|p_{n+1},\dots,p_{n+m}»_c)$$
$$\text{where } n = \mathit{arity}(\texttt{a}) \text{ and } m = \mathit{arity}(\texttt{b})$$

While the Alloy grammar allows multiplicities around the Cartesian product operator, these can only occur when the product expression is used in the specific cases in the grammar. The translation of the multiplicity is handled in those places; see section 5.4.4 for details.

5.3.12. Transpose

The Alloy transpose operator reverses the order of the columns in a relation. It is translated by reversing the order of the predicate list:

$$\mathcal{E}«\texttt{\~{}a}\|p_1,\dots,p_n»_c := \mathcal{E}«\texttt{a}\|p_n,\dots,p_1»_c$$

5.3.13. Domain and Range Restriction

The domain restriction operator `s <: r` restricts the domain of the relation `r` to the set `s`. That is, the domain restriction `s <: r` is the set of tuples from `r` where the first element is contained in the set `s`.

Similarly, the range restriction `r :> s` is the set of tuples from `r` where the last element is contained in the set `s`.

Both operators are translated by materializing the quantifier for the column being restricted:

$$\mathcal{E}«\texttt{s <: r}\|p_1,\dots,p_n»_c := (\texttt{\textbackslash exists}\ \mathcal{T}_1[\texttt{s}]\ \texttt{t;}\ p_1(\texttt{t})\ \texttt{\&\&}\ \mathcal{E}«\texttt{r}\|\mathit{lift}(\texttt{t}),p_2,\dots,p_n»_c$$
$$\texttt{\&\&}\ \mathcal{E}«\texttt{s}\|\mathit{lift}(\texttt{t})»_c)$$
$$\mathcal{E}«\texttt{r >: s}\|p_1,\dots,p_n»_c := (\texttt{\textbackslash exists}\ \mathcal{T}_1[\texttt{s}]\ \texttt{t;}\ p_n(\texttt{t})\ \texttt{\&\&}\ \mathcal{E}«\texttt{r}\|p_1,\dots,p_{n-1},\mathit{lift}(\texttt{t})»_c$$
$$\texttt{\&\&}\ \mathcal{E}«\texttt{s}\|\mathit{lift}(\texttt{t})»_c)$$

5.3.14. Set comprehensions

Alloy allows set comprehension syntax inside relational expressions. The syntax declares variables $\texttt{v}_i$ for each column in the relation being defined. The possible values for each $\texttt{v}_i$ are drawn from the set specified by the relational expression $\texttt{r}_i$, which must produce a unary relation. The relation specified by the comprehension syntax then contains all tuples $\texttt{v}_1\texttt{->}\dots\texttt{->v}_n$ for which the formula `F` is true.

The translation of the set comprehension syntax works by materializing the existential quantifier for the whole tuple:

$$\mathcal{E}«\texttt{\{v}_1\ \texttt{:}\ \texttt{r}_1\texttt{,}\ \dots\texttt{,}\ \texttt{v}_n\ \texttt{:}\ \texttt{r}_n\ \texttt{|}\ \texttt{F \}}\|p_1,\dots,p_n»_c :=$$
$$(\texttt{\textbackslash exists}\ \mathcal{T}_1[\texttt{v}_1]\ \texttt{v}_1\texttt{,}\ \dots\texttt{,}\ \mathcal{T}_1[\texttt{v}_n]\ \texttt{v}_n\texttt{;}\ p_1(\texttt{v}_1)\ \texttt{\&\&}\ \dots\ \texttt{\&\&}\ p_n(\texttt{v}_n)$$
$$\texttt{\&\&}\ \mathcal{E}«\texttt{r}_1\|\mathit{lift}(\texttt{v}_1)»_{c_1}\ \texttt{\&\&}\ \dots\ \texttt{\&\&}\ \mathcal{E}«\texttt{r}_n\|\mathit{lift}(\texttt{v}_n)»_{c_n}\ \texttt{\&\&}\ \mathcal{B}«\texttt{F}»_{c_{n+1}})$$

As each $\texttt{r}_i$ may access the previously declared variables, special care must be taken with the translation context: The context c_i used for the relation $\texttt{r}_i$ is the original translation context c, except that the mapping of Alloy variables to JML variables is extended with all $\texttt{v}_j$ for $j < i$. Finally, the context c_{n+1} for the translation of the formula `F` is the original context c, extended with all the variables $\texttt{v}_i$.

5.3.15. Conditional expression

Alloy conditional expressions are converted into Java conditionals:

$$\mathcal{E}«\texttt{f implies a else b}\|p_1,\ldots,p_n»_c := (\mathcal{B}«\texttt{f}»_c\ \texttt{?}\ \mathcal{E}«\texttt{a}\|p_1,\ldots,p_n»_c\ \texttt{:}\ \mathcal{E}«\texttt{b}\|p_1,\ldots,p_n»_c)$$

An Alloy conditional expression can also be a formula. In that case, it is handled by the translation rules in section 5.4.1.

5.3.16. Cardinality

The cardinality operator `#` is translated to the JML `\num_of` construct.

$$\mathcal{E}«\texttt{\#r}\|p_1»_c = p_1(\texttt{\textbackslash num\_of}\ \mathcal{T}_1[\texttt{r}]\ \texttt{o}_1,\ \ldots,\ \mathcal{T}_n[\texttt{r}]\ \texttt{o}_n;\ \mathcal{E}«\texttt{r}\|\mathit{lift}(\texttt{o}_1),\ldots,\mathit{lift}(\texttt{o}_n)»_c)$$

As defined in the JML reference manual, the `(\num_of T x; P(x))` quantifier is seen as syntax sugar for `(\sum T x; P(x); 1L)`. This means that the simplification rule defined for the quantifier can be applied to the translation of the cardinality operator.

Unfortunately, `\num_of` is not supported by KeY. As such, the cardinality operator is not usable when KeY is used with the Alloy2JML output.

5.3.17. Sum quantifier

The sum quantifier "`sum x: e | i`" computes the sum of all `i` while binding the variable `x` to the values from the set `e`.

The translation uses the JML `\sum` operator, as well as the integer translation function $\mathcal{I}$:

$$\mathcal{E}«\texttt{sum x: e | i}\|p_1»_c = p_1(\texttt{\textbackslash sum}\ \mathcal{T}_1[\texttt{e}]\ \texttt{obj};\ \mathcal{E}«\texttt{e}\|\mathit{lift}(\texttt{obj})»_c;\ \mathcal{I}«\texttt{i}»_{c'})$$

The translation context c' is the translation context c, but extended with the variable mapping from `x` to `obj`.

Alloy sum quantifiers with multiple variables are considered to be syntax sugar for nested sum quantifiers:

$$\mathcal{E}«\texttt{sum x1: e1, x2: e2 | i}\|p_1»_c = \mathcal{E}«\texttt{sum x1 : e1 | (sum x2 : e2 | i)}\|p_1»_c$$

5.4. Translation of formulas

The translation of Alloy formulas is performed using the function $\mathcal{B}«\texttt{F}»_c$.
c is the translation context; while `F` is the input Alloy formula. The result of the $\mathcal{B}$ translation function is a JML boolean expression.

5.4.1. Logical operators

The standard logical operators are directly translated to JML:

$$\begin{aligned}
\mathcal{B}«\texttt{!F}»_c &:= (\texttt{!}\mathcal{B}«\texttt{F}»_c) \\
\mathcal{B}«\texttt{F and G}»_c &:= (\mathcal{B}«\texttt{F}»_c\ \texttt{\&\&}\ \mathcal{B}«\texttt{G}»_c) \\
\mathcal{B}«\texttt{F or G}»_c &:= (\mathcal{B}«\texttt{F}»_c\ \texttt{||}\ \mathcal{B}«\texttt{G}»_c) \\
\mathcal{B}«\texttt{F iff G}»_c &:= (\mathcal{B}«\texttt{F}»_c\ \texttt{<==>}\ \mathcal{B}«\texttt{G}»_c) \\
\mathcal{B}«\texttt{F implies G}»_c &:= (\mathcal{B}«\texttt{F}»_c\ \texttt{==>}\ \mathcal{B}«\texttt{G}»_c) \\
\mathcal{B}«\texttt{F implies G else H}»_c &:= (\mathcal{B}«\texttt{F}»_c\ \texttt{?}\ \mathcal{B}«\texttt{G}»_c\ \texttt{:}\ \mathcal{B}«\texttt{H}»_c)
\end{aligned}$$

5.4.2. Binary relational operators

To check whether two Alloy relations are subsets of each other, or whether they are equal, we need to quantify over the tuples in the relation:

$$\begin{aligned}
\mathcal{B}«\texttt{a in b}»_c :=& \texttt{(\textbackslash forall } \mathcal{T}_1[\texttt{a+b}]\ \texttt{o}_1\texttt{, ..., } \mathcal{T}_n[\texttt{a+b}]\ \texttt{o}_n\texttt{;}\\
& \mathcal{E}«\texttt{a}\|\mathit{lift}(\texttt{o}_1),\dots,\mathit{lift}(\texttt{o}_n)»_c \texttt{ ==> } \mathcal{E}«\texttt{b}\|\mathit{lift}(\texttt{o}_1),\dots,\mathit{lift}(\texttt{o}_n)»_c\texttt{)}\\
\mathcal{B}«\texttt{a = b}»_c :=& \texttt{(\textbackslash forall } \mathcal{T}_1[\texttt{a+b}]\ \texttt{o}_1\texttt{, ..., } \mathcal{T}_n[\texttt{a+b}]\ \texttt{o}_n\texttt{;}\\
& \mathcal{E}«\texttt{a}\|\mathit{lift}(\texttt{o}_1),\dots,\mathit{lift}(\texttt{o}_n)»_c \texttt{ <==> } \mathcal{E}«\texttt{b}\|\mathit{lift}(\texttt{o}_1),\dots,\mathit{lift}(\texttt{o}_n)»_c\texttt{)}
\end{aligned}$$

We use Alloy set union with the type function $\mathcal{T}$ to determine the common base type of the $\mathcal{T}_i[\texttt{a}]$ and $\mathcal{T}_i[\texttt{b}]$.

5.4.3. Quantified expressions

When a multiplicity is used as an unary operator with a relational expression `r`, it produces a Boolean based on the number of entries in the relation.

$$\begin{aligned}
\mathcal{B}«\texttt{no r}»_c :=& \texttt{(!}\mathcal{E}«\texttt{r}\| \underbrace{\mathit{nonnull}}_{\mathit{arity}(\texttt{r})\text{ times}} »_c\texttt{)}\\
\mathcal{B}«\texttt{some r}»_c :=& \mathcal{E}«\texttt{r}\| \underbrace{\mathit{nonnull}}_{\mathit{arity}(\texttt{r})\text{ times}} »_c\\
\mathcal{B}«\texttt{lone r}»_c :=& \texttt{(\textbackslash forall } \mathcal{T}_1[\texttt{r}]\ \texttt{a}_1\texttt{, ..., } \mathcal{T}_n[\texttt{r}]\ \texttt{a}_n\texttt{, } \mathcal{T}_1[\texttt{r}]\ \texttt{b}_1\texttt{, ..., } \mathcal{T}_n[\texttt{r}]\ \texttt{b}_n\texttt{;}\\
& \texttt{(}\mathcal{E}«\texttt{r}\|\mathit{lift}(\texttt{a}_1),\dots,\mathit{lift}(\texttt{a}_n)»_c \texttt{ \&\& } \mathcal{E}«\texttt{r}\|\mathit{lift}(\texttt{b}_1),\dots,\mathit{lift}(\texttt{b}_n)»_c\texttt{)}\\
& \texttt{==> (a}_1\texttt{ == b}_1\texttt{ \&\& ... \&\& a}_n\texttt{ == b}_n\texttt{))}\\
\mathcal{B}«\texttt{one r}»_c :=& \texttt{(}\mathcal{B}«\texttt{some r}»_c \texttt{ \&\& } \mathcal{B}«\texttt{lone r}»_c\texttt{)}
\end{aligned}$$

The `no` and `some` translations use the *nonnull* predicate to check for the emptyness/non-emptyness of the relation `r`. The `lone` translation quantifies over two tuples `a` and `b` that are elements of the relation `r`, and asserts that those tuples must be the same.

5.4.4. Multiplicity Constraints

Alloy allows occurrences of multiplicities in expressions only on the right-hand side of the subset operator `in`; and as part of the range expression for variables introduced in quantifiers. In the latter case, multiplicities other than `one` cause higher-order quantification which is not supported by our translation. This means that our translation only needs to handle multiplicities on the right-hand-side of the subset operator.

For unary relations, the syntax is "`a in` *m* `b`", where *m* is a multiplicity (`lone`, `some` or `one`). The multiplicity constraint applies to the relation `a`, and can be translated as a quantified expression separately from the subset operator:

$$\mathcal{B}«\texttt{a in } m \texttt{ b}»_c := \texttt{(}\mathcal{B}«\texttt{a in b}»_c \texttt{ \&\& } \mathcal{B}«m \texttt{ a}»_c\texttt{)}$$

Multiplicity constraints on higher-arity relations are also treated as syntax sugar, using the same approach as [Gei11, 3.2]:

> The second form of multiplicity annotations is an annotated product operator `->`. For multiplicity keywords n and m, the multiplicity restriction of `r in A` *n* `->` *m* `B` can be expressed by the following formulas:
>
> `all a: A |` *m* `a.r`
>
> `all b: B |` *n* `r.b`
>
> Annotated product operations may also be nested. These can be desugared in a similar way, using consecutive join operations. For example, the multiplicity restriction of `r in A -> one B -> lone C` is expressed by
>
> `(all a: A | one a.r) && (all a: A, b: B | lone b.(a.r))`

5.4.5. Quantified formulas

The universal and existential quantifiers in Alloy are translated to the corresponding JML quantifiers:

$$\mathcal{B}«\texttt{all v : a | F}»_c := \texttt{(\textbackslash forall } \mathcal{T}_1[\text{a}] \texttt{ obj; } \mathcal{E}«\text{a}\|\textit{lift}(\texttt{obj})»_c \texttt{ ==> } \mathcal{B}«\texttt{F}»_{c^*}\texttt{)}$$
$$\mathcal{B}«\texttt{some v : a | F}»_c := \texttt{(\textbackslash exists } \mathcal{T}_1[\text{a}] \texttt{ obj; } \mathcal{E}«\text{a}\|\textit{lift}(\texttt{obj})»_c \texttt{ \&\& } \mathcal{B}«\texttt{F}»_{c^*}\texttt{)}$$
$$\mathcal{B}«\texttt{no v : a | F}»_c := \texttt{(\textbackslash forall } \mathcal{T}_1[\text{a}] \texttt{ obj; } \mathcal{E}«\text{a}\|\textit{lift}(\texttt{obj})»_c \texttt{ ==> !} \mathcal{B}«\texttt{F}»_{c^*}\texttt{))}$$

The translation context c^* is the translation context c, except that the mapping of Alloy variables to JML variables is extended with the mapping from `v` to `obj`.

Quantified formulas involving multiple variables are desugared into nested usage of quantified formulas:

$$\mathcal{B}«\texttt{all v1 : a1, v2 : a2 | F}»_c := \mathcal{B}«\texttt{all v1 : a1 | (all v2 : a2 | F)}»_c$$
$$\mathcal{B}«\texttt{some v1 : a1, v2 : a2 | F}»_c := \mathcal{B}«\texttt{some v1 : a1 | (some v2 : a2 | F)}»_c$$
$$\mathcal{B}«\texttt{no v1 : a1, v2 : a2 | F}»_c := \mathcal{B}«\texttt{no v1 : a1 | (some v2 : a2 | F)}»_c$$

Alloy allows multiplicity constraints on variable declarations, including those in quantified formulas. However, the Alloy analyzer does not support the higher-order quantification that is necessary when a bound variable refers to a set or relation instead of a single atom. Our translation to JML has the same restriction, and will only accept the default multiplicity `one` in quantified formulas.

The `one` and `lone` quantified formulas are treated as syntax sugar for the `one` and `lone` quantified expressions in combination with set comprehensions:

$$\mathcal{B}«\texttt{lone v : a | F}»_c := \mathcal{B}«\texttt{lone \{v : a | F\}}»_c$$
$$\mathcal{B}«\texttt{one v : a | F}»_c := \mathcal{B}«\texttt{one \{v : a | F\}}»_c$$

Quantified formulas involving multiple variables are translated using set comprehensions with multiple variables.

5.4.6. Integer comparisons

The arithmetic comparison operators in Alloy accept sets of integer as their arguments. If a non-scalar set is supplied, the comparison acts on the sum of the values in the set. [Jac06, p. 290]

We define a translation function $\mathcal{I}«\texttt{r}»_c$ that takes an Alloy relational expression `r` that represents a set of integers, and returns a JML expression that evaluates to the sum of the integers in the set.

Unlike $\mathcal{E}$ and $\mathcal{B}$, this translation function is not defined recursively over the Alloy syntax tree. Instead, it always uses the JML `\sum` quantifier with $\mathcal{E}$ function:

$$\mathcal{I}«\texttt{r}»_c := \texttt{(\textbackslash sum int i; } \mathcal{E}«\texttt{r}\|\textit{lift}(\texttt{i})»_c\texttt{; i)}$$

As usual, `i` stands for a newly created unique identifier in order to avoid name collisions.

This leads us to the following definition of the comparison operators:

$$\mathcal{B}«\texttt{a < b}»_c := (\mathcal{I}«\text{a}»_c \texttt{ < } \mathcal{I}«\text{b}»_c)$$
$$\mathcal{B}«\texttt{a > b}»_c := (\mathcal{I}«\text{a}»_c \texttt{ > } \mathcal{I}«\text{b}»_c)$$
$$\mathcal{B}«\texttt{a =< b}»_c := (\mathcal{I}«\text{a}»_c \texttt{ <= } \mathcal{I}«\text{b}»_c)$$
$$\mathcal{B}«\texttt{a >= b}»_c := (\mathcal{I}«\text{a}»_c \texttt{ >= } \mathcal{I}«\text{b}»_c)$$

Like the existential quantifiers in the definition of $\mathcal{E}$, the `\sum` operator can often be eliminated by the simplification step following the translation. In particular, if the Alloy expression is a variable, or a field access on a variable, the `\sum` operator can always be eliminated.

This simplification is critical when using the KeY proving tool: KeY only supports the `\sum` operator when explicit bounds are provided in the form `(\sum int i; i <= lowerBound && i < upperBound; i)`. This is almost never the case with the output of the translation function[1], so integer comparisons must be simplified to be usable with KeY.

It should be noted that it is possible to specify unbounded sums in Alloy. For example, the expression `val =< Int` asserts that `val` is less or equal than the sum of all integers.

In this example, the Alloy Analyzer finds one model for `val`: the integer -8. This is because the Alloy Analyzer performs bounded analysis and uses 4-bit integers by default, with values from -8 to $+7$. In JML, the quantification uses Java integers, which too are limited (to 32 bits). This means that, at least in theory, such expressions have a well-defined value. However, as the value is differs between Alloy and JML (and is nonsensical in both), unbounded sums should be avoided.

5.5. Simplifications

The translation function $\mathcal{E}$ introduces many quantifiers that are often redundant. As an example, we will consider the translation of the expression "`no x.left`", where `x` is a method parameter of type `Tree` that is known to be non-null (has no JML `nullable` annotation):

$$\begin{aligned}
\mathcal{B}«\texttt{no x.left}»_c &= (\texttt{!}\mathcal{E}«\texttt{x.left}\|\mathit{nonnull}»_c) \\
&= (\texttt{!(\textbackslash exists Tree t; } \mathcal{E}«\texttt{x}\|\mathit{lift}(\texttt{t})»_c \texttt{ \&\& } \mathcal{E}«\texttt{left}\|\mathit{lift}(\texttt{t}), \mathit{nonnull}»_c)) \\
&= (\texttt{!(\textbackslash exists Tree t; } \mathit{lift}(\texttt{t})(\texttt{x}) \\
&\quad \texttt{\&\& (\textbackslash exists Tree obj; } \mathit{lift}(\texttt{t})(\texttt{obj}) \texttt{ \&\& } \mathit{nonnull}(\texttt{obj.member})))) \\
&= (\texttt{!(\textbackslash exists Tree t; x == t} \\
&\quad \texttt{\&\& (\textbackslash exists Tree obj; obj == t \&\& obj.member != null)))}
\end{aligned}$$

Both quantifiers are redundant: they can only be true if the newly introduced variable is equal to an existing variable.

5.5.1. Eliminating existential quantifiers

A simplification rule can detect this case and eliminate the quantifier:

$$\begin{aligned}
&\texttt{(\textbackslash exists T obj; } f_1 \texttt{ \&\& } x \texttt{ == obj \&\& } f_2(\texttt{obj})) \\
&\quad \hookrightarrow (f_1 \texttt{ \&\& } x \texttt{ instanceof T \&\& } f_2((\texttt{T})x))
\end{aligned} \tag{5.1}$$

f_1 and x are arbitrary JML expressions that do not use the `obj` variable. f_2 is an arbitrary JML expression that may use the `obj` variable.

Special care must be taken if f_2 uses `obj` within an `\old(...)` expression: moving the expression x into this context may change its semantics. This happens if x accesses the post-state in some way (contains field access or quantifier). Thus, this simplification rule is not applied (and the existential quantifier preserved) if x contains such accesses to the post-state, and f_2 uses `obj` within an `\old(...)` expression. This way, we keep quantifiers

[1]The translation may match this pattern if the Alloy specification contains such bounds, for example "`{i:Int | i =< 0 and i < 5} = 10`".

that are necessary due to the inflexibility of the JML `\old` keyword as described in chapter 3.3.2.

The simplified expression for the quantifiers contains an `instanceof` operator and a cast to ensure that the simplified expression is equivalent even if the type of expression x differs from `T`.

5.5.2. Eliminating redundant casts

Casts introduced by the simplification rule above are often redundant. Such redundant casts are eliminated where possible by additional simplification rules:

$$\begin{aligned} (x \texttt{ instanceof T}) &\hookrightarrow (x \texttt{ != null}) \qquad (5.2)\\ &\quad \text{if the type of } x \text{ is a subtype of } \texttt{T}\\ ((\texttt{T})x) &\hookrightarrow (x) \qquad (5.3)\\ &\quad \text{if the type of } x \text{ is equal to } \texttt{T} \end{aligned}$$

Null checks introduced by this rule are often redundant. If x is statically known to be non-null, the null check is eliminated:

$$(x \texttt{ != null}) \hookrightarrow \texttt{true} \qquad (5.4)$$

Statically known to be non-null are:

1. bound variables
2. method parameters, unless they a marked with the JML `nullable` modifier
3. fields, unless they a marked with the JML `nullable` modifier

Applying these simplifications to our example:

$$\begin{aligned} \mathcal{B}«\texttt{no x.left}»_c &= \texttt{(!(\textbackslash exists Tree t; x == t}\\ &\quad \texttt{\&\& (\textbackslash exists Tree obj; obj == t \&\& obj.left != null)))}\\ &\overset{5.1}{=} \texttt{(!(\textbackslash exists Tree t; x == t}\\ &\quad \texttt{\&\& (t instanceof Tree \&\& ((Tree)t).left != null)))}\\ &\overset{5.2}{=} \texttt{(!(\textbackslash exists Tree t; x == t}\\ &\quad \texttt{\&\& (t != null \&\& t.left != null)))}\\ &\overset{5.4}{=} \texttt{(!(\textbackslash exists Tree t; x == t \&\& t.left != null))}\\ &\overset{5.1}{=} \texttt{(!(x instanceof Tree \&\& ((Tree)x).left != null))}\\ &\overset{5.2}{=} \texttt{!(x != null \&\& x.left != null)}\\ &\overset{5.4}{=} \texttt{x.left == null} \end{aligned}$$

In general, whenever the translation function is used on a field signature with a *lift*() predicate, and the types are correct so that the cast can be eliminated, the simplifications described so far will allow us to simply invoke the second predicate on the field value:

$$\mathcal{E}«\texttt{member}\|\mathit{lift}(\texttt{obj}), p_2»_c = p_2(\texttt{obj.member})$$

The null check simplification 5.4 can always be used here: the usage of *lift*(`obj`) as a predicate implies that `obj` is statically known to be non-null.

5.5.3. Eliminating redundant quantifiers

In addition to the elimination of the existential quantifier, we also need a simplification rule to eliminate universal quantifiers. These occur in the translation of the `in` operator.

$$\begin{aligned}&\texttt{(\textbackslash forall T obj; } f_1 \texttt{ \&\& } x \texttt{ == obj ==> } f_2\texttt{(obj))}\\ &\quad\hookrightarrow \texttt{(}f_1 \texttt{ \&\& } x \texttt{ instanceof T ==> } f_2\texttt{((T)}x\texttt{))}\end{aligned} \tag{5.5}$$

f_1, x and f_2 are JML expressions as in the rule for eliminating the existential quantifier (5.1); and the same restriction regarding JML `\old(...)` applies.

The third related simplification rule is the elimination of the `\sum` operator that occurs in the translation of the integer comparisons:

$$\begin{aligned}&\texttt{(\textbackslash sum T obj; } f_1 \texttt{ \&\& } x \texttt{ == obj \&\& } f_2\texttt{(obj); } f_3\texttt{(obj))}\\ &\hookrightarrow \texttt{((}f_1 \texttt{ \&\& } x \texttt{ instanceof T \&\& } f_2(x)\texttt{) ? } f_3(x) \texttt{ : 0)}\end{aligned} \tag{5.6}$$

Again, f_1, x and f_2 are defined as before, and the `\old(...)` restriction applies. f_3 is an additional JML expression that may access `obj`. Because the `\num_of` quantifier is defined as syntax sugar for `\sum`, this simplification rule applies to `\num_of` as well.

A similar rule is useful for simplifying the result of set equality comparisons:

$$\begin{aligned}&\texttt{(\textbackslash forall T obj; } f_1 \texttt{ \&\& } x \texttt{ == obj <==> } f_2 \texttt{ \&\& } y \texttt{ == obj)}\\ &\qquad\hookrightarrow \texttt{((}f_1 \texttt{ ? } x \texttt{ : null) == (}f_2 \texttt{ ? } y \texttt{ : null))}\end{aligned} \tag{5.7}$$

In this rule, f_1, f_2, x and y are arbitrary JML expressions that do not access `obj`. The types of x and y also must be subtypes of `T`, and may not be primitive types.

For example, when `x` and `y` are parameters of type `nullable Node`:

$$\begin{aligned}\mathcal{B}\langle\!\langle\texttt{x.next = y}\rangle\!\rangle_C &= \texttt{\textbackslash forall Node obj; x != null \&\& x.next == obj <==> y == obj}\\ &\overset{5.7}{=} \texttt{(x != null ? x.next : null) == y}\end{aligned}$$

For the primitive types `int` and `boolean`, the rule above cannot be used as these types do not allow the use of a null reference to indicate a missing value. Instead, we use the following rule:

$$\begin{aligned}&\texttt{(\textbackslash forall T obj; } f_1 \texttt{ \&\& } x \texttt{ == obj <==> } f_2 \texttt{ \&\& } y \texttt{ == obj)}\\ &\qquad\hookrightarrow \texttt{((}f_1 \texttt{ \&\& } f_2 \texttt{ \&\& } x \texttt{ == } y\texttt{) || (!}f_1 \texttt{ \&\& !}f_2\texttt{))}\end{aligned} \tag{5.8}$$

Simplification of `\fresh`:

$$\texttt{\textbackslash fresh(}x\texttt{)} \hookrightarrow \texttt{false} \tag{5.9}$$

if the expression x does not access to the post-state, does not contain `\result`, and does not contain any free non-program variables, it is impossible for the expression to refer to a newly created object.

5.6. Translation of Contract clauses

The `requires`, `ensures` and `invariant` clauses all are followed by an Alloy formula. To translate these clauses into the equivalent JML clauses, the $\mathcal{B}$ function is used to translate the formula into a Boolean JML expression.

In all cases, the initial translation context passed to the $\mathcal{B}$ function:

- contains all Alloy predicates declared in the input file
- maps the Alloy variables for the method parameters to the actual Java method parameters

Additionally, in the case of postconditions (`ensures` clause), the initial translation context maps the Alloy `result` variable to the JML `\result` variable. The context is also marked as "within a postcondition", which causes `\old(...)` expressions to be generated around accesses to the pre-state. As a special case, methods marked as `pure` do not have this flag set. This prevents the translation from creating unnecessary `\old(...)` expressions, as the pre-state is equal to the post-state within pure methods.

5.6.1. Modifies clause

The `modifies` clause specifies a comma-separated list of heap locations that may be modified by the method. Each location is given by the syntax "*alloy-expression . identifier*". The identifier refers to the Java field that may be modified, and the Alloy expression specifies the set of target object instances.

If the target expression is simple (for example, "`this`", or another non-nullable parameter), it can be directly translated into JML. For example, the clause

```
//$ modifies this.length
```

can be directly translated into the corresponding JML clause

```
//@ modifies this.length;
```

However, in the general case, the target object set cannot be expressed as a JML expression. In this case, Alloy2JML will generate a JML modifies clause that is less specific: we allow the modification on the field on all object instances.

In JML* for KeY, this is done using `\infinite_union`:

```
//@ modifies \infinite_union(List node; node.next);
```

In JML, a static data group could be used for the same purpose.

Additionally, we generate a postcondition that specifies that the field value is unchanged on any objects not in the target object set:

```
ensures all obj : DeclaringType - fresh - target | obj.field' = obj.field
```

This postcondition is translated to JML as usual using the $\mathcal{B}$ function.

5.6.2. Predicates

Alloy predicates in the input file are converted into static model methods.

For example:

```
class LinkedList {
    ...
    int length;

    //$ pred isEmpty[list : LinkedList] {
    //$   list.length = 0
    //$ }

    //$ ensures this.isEmpty'
    public void clear() { ... }
}
```

Translated to JML, we get:

```
class LinkedList {
    ...
    int length;

    /*@ public normal_behavior
        ensures \result <==> list != null && list.length == 0;
        static model helper strictly_pure
        boolean isEmpty(nullable LinkedList l);
    @*/

    //@ ensures isEmpty(this);
    public void clear() { ... }
}
```

The Alloy expression passed to a predicate must be a simple expression: a program variable, the built-in `none` set, or a field access on another simple expression. This restriction exists because the generated model method only accepts a single object instance, not an arbitrary set. If `none` is passed, or the program variable or any field involved is null, the caller will pass a null reference to the model method. This corresponds to an empty set in Alloy semantics.

The model method is always specified as in the example above; only the right-hand side of the equivalence operator is translated from the predicate body:
`ensures \result <==>` $\mathcal{B}$«`body`»$_c$
The necessary null checks are added by the simplification step (quantifier elimination followed by rule 5.2).

6. Translation of transitive closure

The Alloy operator for the transitive closure "`^r`" can be applied to any binary relation. In Alloy, it is defined as a union of relational joins:

```
  ^r = r + r.r + r.r.r + ...
```

Because the Alloy analyzer works within finite bounds, this definition is finite, and thus allows the transitive closure to be represented in first-order logic.

Additionally, Alloy defines the reflexive transitive closure `*r`:

```
  *r = iden + r + r.r + r.r.r + ...
```

It should be noted that `*r` is reflexive on the whole universe, and thus always has the type `univ -> univ`.

To implement a transitive closure in JML, the `\reach` expression looks like a potential candidate. In JML, `\reach(x)` is defined to return the smallest `JMLObjectSet` that contains `x` and all objects reachable from `x`. When `x` is a data group, `\reach(x)` only includes the objects reachable through fields in that data group, thus allowing control over the fields involved in the transitive closure.

However, there are several problems with using `\reach` in this way:

- `\reach` always computes the reflexive transitive closure; there is no corresponding JML construct for a non-reflexive transitive closure.
- While data groups may be used to specify a set of fields; `\reach` cannot be used for the transitive closure of more complex relations. For example, Alloy "`node.*(~next)`" refers the set of predecessors of a given node.
- KeY does not support the standard JML `\reach` predicate with data groups, and instead uses its own syntax.

We instead use model methods to define our own reachability predicate. This is based on an approach developed by Gladisch and Tyszberowicz [GT13].

6.1. Linked lists

In the simple case where the transitive closure is directly applied to a field signature (`*next`), the transitive closure contains all nodes in the singly-linked list formed by the field `next`. Here, we can use a `getNext()` method to retrieve nodes from the list:

```
/*@ normal_behavior
    ensures (n < 0 ==> \result == null);
    ensures (n == 0 ==> \result == head);
    ensures (n > 0 ==> \result == (
                   head != null ? getNext(head.next, n-1) : null));
    ensures (n > 0 ==> \result == (
                   getNext(head, n-1) != null ? getNext(head, n-1).next : null));
    accessible \infinite_union(Entry e; e.next);
    measured_by n;
@*/
public static /*@ helper strictly_pure nullable @*/ Entry getNext(/*@ nullable @
    */ Entry head, int n)
{ ... }
```

The `accessible` clause is using the JML* construct "`\infinite_union`" to specify that the value of the `getNext()` method depends only on the `next` field. In regular JML, the same `accessible` clause could be specified using a static `JMLDataGroup` that contains the `next` field of all `Entry` instances. However, KeY currently does not support data groups, which forces us to use the equivalent JML* syntax. The `accessible` clause is an over-approximation: the value does not actually depend on all `Entry` instances, but only those in the transitive closure. We use this over-approximation as we cannot specify the transitive closure in the `accessible` clause in terms of the `getNext()` method, and would require another way to specify the transitive closure.

Using the `getNext()` method, we can obtain the transitive closure over the `next` field by quantifying over the number of steps:

$$\mathcal{E}\langle\!\langle \texttt{\^{}next} \| p_1, p_2 \rangle\!\rangle_c := (\texttt{\textbackslash exists}\ \mathcal{T}_1[\texttt{next}]\ \texttt{obj;}\ p_1(\texttt{obj})\ \texttt{\&\&}$$
$$(\texttt{\textbackslash exists int steps; steps > 0;}\ p_2(\texttt{getNext(obj, steps)})))$$

For example, when checking whether a given node is contained in the linked list starting at `head`, most of the quantifiers can be eliminated by the simplification, and the translation looks like this:

$$\mathcal{B}\langle\!\langle \texttt{node in head.\^{}next} \rangle\!\rangle_c =$$
$$(\texttt{\textbackslash exists int steps; steps > 0; getNext(head, steps) == node})$$

The usage of an explicit number of steps may cause problems for automatic theorem provers: they may be unable to find the correct instantiation for the number of steps. However, it can also be of an advantage if the number of steps plays an explicit role in the program, for example in a linked list implementation where methods access list items by index.

In fact, treating a linked list as a sequence that can be indexed is so useful that we expose this functionality to our input Alloy specifications. We do this by defining a function "`pow[r, steps]`" that joins the relation `r` with itself `steps` times.

Using this function, we can specify the behavior of a linked list's `get()` method:

```
//$ requires i >= 0
//$ ensures result = head.(pow[next,i]).data
public Object get(int i) { ... }
```

This allows Alloy specifications to talk about explicit numbers of steps, while simultaneously allowing easy access to the set of all list elements using the transitive closure.

6.2. Generalizing the transitive closure

However, this approach to the transitive closure only works for simple fields, as they can be treated as a singly-linked list (potentially with cycles). In general, an arbitrary relation can be interpreted as a directed graph, with the transitive closure being the set of nodes that are reachable in any number of steps. Any approach using a `get()`-style method would have to take an argument that specifies a path through the graph.

For a simple union of several fields, like the transitive closure "`^(left+right)`" that might get used in the specification of a binary tree, the path could be encoded within an integer argument:

```
getNode(obj, 1) = obj
getNode(obj, 2*i) = getNode(obj, i).left
getNode(obj, 2*i+1) = getNode(obj, i).right
```

With this definition, an inductive proof over the graph structure would require complete induction, or a custom induction rule. Additionally, it is unclear how this would work with arbitrary Alloy relations. Consider a graph where nodes contain a dynamically-sized array of child nodes: Every single step can reach an arbitrary number of nodes, so that we would need to use a list of integers to represent a path. Forming the transitive closure would then require quantifying over all possible lists. For more complicated Alloy relations (for example, one constructed using a set comprehension), it is even less clear how paths could be represented.

So, instead of providing a `get(start, i)` query method that returns the node after *i* steps, we will use a boolean query method that returns whether a node can be found after a given number of steps. For the binary tree example, this method is specified as follows:

```
/*@
  public normal_behavior
  ensures steps < 0 ==> \result == false;
  ensures steps == 0 ==> (\result <==> root == node && root != null);
  ensures steps > 0
    ==> (\result <==> root != null && (   hasNode(root.left,  steps - 1, node)
                                       || hasNode(root.right, steps - 1, node)));
  ensures steps > 0 ==> (\result <==> node != null &&
      (\exists Tree parent; hasNode(root, steps - 1, parent)
                            && (parent.left == node || parent.right == node)));
  measured_by steps;
  static model helper strictly_pure
  boolean hasNode(nullable Tree root, int steps, nullable Tree node);
@*/
```

This method returns `true` if, starting from the node `root`, the node `entry` can be reached in exactly the given number of steps. Otherwise, the method returns `false`. In case of invalid input (negative number of steps, or `null` reference for one the tree nodes), the method always returns `false`.

Note the two postconditions for the `steps > 0` case: both a head-recursive and a tail-recursive specification are available. Both are equivalent and either can be proven from the other by induction over the number of steps. As both kinds of recursion may be necessary to verify a given program (for example, see `contains_loop` in chapter 8.2.2), our translation will generate both the head-recursive and the tail-recursive specifications.

The recursive specifications split the path through the graph of length `steps` into a path of length `steps-1` (which is handled by the recursive `hasNode()` call) and a single step (at either the beginning or the end of the longer path). This single step can be handled by our existing translation function. Therefore, the `hasNode()` approach can be generalized to translate the transitive closure of almost any Alloy relational expression supported by our translation.

6.3. Query method definition

Given an Alloy relational expression `r`, we define the query method $Q_{\texttt{r}}$ to be the `hasNode()`-style method where the recursive specification (the "single step") is given by the relation `r`:

```
/*@
public normal_behavior
ensures steps < 0 ==> \result == false;
ensures steps == 0 ==> (\result <==> root == node && root != null);
ensures steps > 0 ==> (\result <==> root != null && $\mathcal{E}$«r||$lift$(root), $headrec_{\texttt{r}}$»$_c$);
ensures steps > 0 ==> (\result <==> node != null && $\mathcal{E}$«r||$tailrec_{\texttt{r}}$, $lift$(node)»$_c$);
measured_by steps;
static model helper strictly_pure
boolean $Q_{\texttt{r}}$(nullable $T$ root, int steps, nullable $T$ node);
@*/
```

In this definition, the type T is the type of the transitive closure as determined by Alloy's type inference – in symbols, $\mathcal{T}_1[\texttt{\^{}r}]$.

headrec$_{\texttt{r}}$ and *tailrec*$_{\texttt{r}}$ are translation predicates that recursively call the method $Q_{\texttt{r}}$:

$$headrec_{\texttt{r}}(e) := \texttt{(}e\texttt{ instanceof }T\texttt{ \&\& }Q_{\texttt{r}}\texttt{((}T\texttt{)}e\texttt{, steps-1, node))}$$

$$tailrec_{\texttt{r}}(e) := \texttt{(}e\texttt{ instanceof }T\texttt{ \&\& }Q_{\texttt{r}}\texttt{(root, steps-1, (}T\texttt{)}e\texttt{))}$$

In the head-recursive ensures clause, the $\mathcal{E}$ function is used to produce an expression that evaluates to `true` if the relation `r` contains a pair $(\texttt{root}, e)$ where `node` is reachable from e in `steps-1` steps. Similarly, the tail-recursive postcondition uses $\mathcal{E}$ to create a JML expression that signifies whether the relation `r` contains a pair $(e, \texttt{node})$ so that e is reachable from `root` in `steps-1` steps.

Overall, the $Q_{\texttt{r}}$ specification works the same way as the `hasNode` specification: considering the graph where objects are nodes and the set of edges is given by the relation `r`, the call $Q_{\texttt{r}}$`(a,n,b)` returns `true` if and only if `b` is reachable from `a` in exactly `n` steps.

It should be noted that the translation predicates involved are well-formed: the explicit null-checks ensure that the *lift*() predicates are well-formed. For the *headrec*$_{\texttt{r}}$ and *tailrec*$_{\texttt{r}}$ predicates, explicit type checks are used to ensure that the method call is possible. The `instanceof` operator also ensures the translation predicates return `false` when given a null reference, although this is not strictly necessary as the recursive $Q_{\texttt{r}}$ call is specified to return `false` on null references.

The last rule for well-formed translation predicates is that they must have the same semantics when they are evaluated within an `\old` expression as they have outside of such an expression. This is trivially guaranteed because the method $Q_{\texttt{r}}$ is `pure`, so that the pre-state and post-state are the same.

Using the $Q_{\texttt{r}}$ definition, we can translate the transitive closure as follows:

$$\mathcal{E}\text{«}\texttt{\^{}r}\|p_1, p_2\text{»}_c := \texttt{(\textbackslash exists }\mathcal{T}_1[\texttt{\^{}r}]\texttt{ obj1, obj2; }p_1\texttt{(obj1) \&\& }p_2\texttt{(obj2) \&\&}$$
$$\texttt{(\textbackslash exists int steps; steps > 0; }Q_{\texttt{r}}\texttt{(obj1, steps, obj2)))}$$

One problem with the definition as described above is that the query method does not have access to the variables in the context of its call. This would cause the translation to fail if the Alloy relation `r` accesses one of those variables. To solve this problem, we pass such variables to the query method as additional parameters.

Consider a specification that takes the transitive closure not over the current `left` and `right` relations, but also adds an edge from node `a` to node `b`, where `a` and `b` are both program variables. This can be specified in Alloy using the following expression:

```
^(left + right + (a -> b))
```

When translated to JML, the parameter list of the query method will be:

```
boolean query(Tree root, int steps, Tree node, Tree a, Tree b)
```

Finally, we disallow taking the transitive closure (or using the `pow` function) of a relation that accesses both the pre- and post-state. This is necessary as it is impossible to pass multiple heap states to a JML model method. If the transitive closure accesses only the pre-state, we use the `\old` operator around the call to the query. In fact, we need to place the `\old` operator around the quantifiers as follows:

$\mathcal{E}«\texttt{\^{}r}\|p_1,p_2»_c :=$ `(\old(\exists` $\mathcal{T}_1$`[^r] obj1, obj2;` p_1`(obj1) &&` p_2`(obj2) &&`
`(\exists int steps; steps > 0;` $Q_{\texttt{r}}$`(obj1, steps, obj2))))`

The reasoning for this quantifier placement is the same as for field accesses (see 5.3.4): If we were to pass objects that might be freshly created in the post-state to the query method, that might cause it to access field values of those objects in the pre-state. Such field accesses would not be well-defined behavior.

6.4. Reflexive closure

Now we will consider the translation of the reflexive transitive closure, `*r`. A simple adjustment to the translation would be to change the requirement on the step count from `steps > 0` to `steps >= 0`. Indeed, in many contexts where the transitive closure is used, this change is sufficient. However, it does not fully capture the semantics of the Alloy `*` operator.

In Alloy, `*r` is defined to be `iden + ^r`. That is, the resulting relation is reflexive over the whole Alloy universe, not just the domain of relation `r`. As our query method uses strongly-typed parameters, it cannot accept these additional elements. An easy fix to this problem would be to treat `*r` as syntax sugar for `iden + ^r`. However, this means the special case for `iden` is always generated, even when doing so is not necessary. Frequently, the transitive closure is immediately joined with a head node, for example in:

```
all n : this.*(left+right) | n.value = 0.
```

Because `this` already is a tree node, the translation of the transitive closure with `steps >= 0` is sufficient, and it does not make sense to introduce a special case for the unused larger domain of the reflexive closure. For this reason, we chose the following translation for the reflexive closure:

$\mathcal{E}«\texttt{*r}\|p_1,p_2»_c :=$ `(`
    `(\exists Object obj;` p_1`(obj) &&` p_2`(obj) && !(obj instanceof` $\mathcal{T}_1$`[^r]))`
  `||`
    `(\exists` $\mathcal{T}_1$`[^r] obj1, obj2;` p_1`(obj1) &&` p_2`(obj2) &&`
      `(\exists int steps; steps >= 0;` $Q_{\texttt{r}}$`(obj1, steps, obj2)))`
  `)`

The first part of the disjunction corresponds to the translation of `iden`, except that it excludes all objects that are supported by the query method and are handled in the second part of the disjunction.

This translation may look more complicated, but it works well with the existing simplification rules. If either p_1 or p_2 is a *lift*(...)-style predicate, the quantifier for `obj` can be eliminated. This causes the expression `obj` to be replaced with the lifted expression, which usually has a more specific type. If this type is a subtype of the type $\mathcal{T}_1$[`^r`], this allows the simplification rule 5.2 to replace the `instanceof` check with the constant `true`. Further simplification based on the use of constants in logical operators will then eliminate the disjunction, leaving only the second part.

This means that for the vast majority of use cases, the translation output of the reflexive transitive closure `*r` is no more complicated than that of the transitive closure `^r`.

6.5. pow[] function

Finally, let us consider the translation of the `pow[]` function that we mentioned at the end of section 6.1. The Alloy semantics of this function are given by the following definition:

```
fun pow[r : univ -> univ, steps:set Int] : univ->univ {
      (0 in steps implies iden   else none->none)
    + (1 in steps implies r      else none->none)
    + (2 in steps implies r.r    else none->none)
    + (3 in steps implies r.r.r  else none->none)
   // ... continue up to the max. integer
}
```

That is, the `pow` function joins the relation `r` with itself `steps` times. The `steps` parameter may be a set of step counts; in this case the function returns the union of its results for the individual step counts.

In fact, the `pow` function can be considered to be a generalized version of the transitive closure:

```
  *r = pow[r, Int]
  ^r = pow[r, Int-0]
```

In Alloy, the function is implemented by exhaustively listing the possible number of steps. In the translation to JML, we translate `pow[r, steps]` similarly to the reflexive closure:

```
        𝓔«pow[r, steps]‖$p_1, p_2$»$_c$ := (
            (\exists Object obj; $p_1$(obj) && $p_2$(obj) && 𝓔«steps‖$lift$(0)»$_c$
                                 && !(obj instanceof $\mathcal{T}_1$[^r]))
          ||
            (\exists $\mathcal{T}_1$[^r] obj1, obj2; $p_1$(obj1) && $p_2$(obj2) &&
              (\exists int num; 𝓔«steps‖$lift$(num)»$_c$ && $Q_{\mathtt{r}}$(obj1, num, obj2)))
          )
```

As with the reflexive closure, the simplification step will usually dramatically simplify the translation output of `pow`.

7. Correctness proof of the translation

We use the Isabelle/HOL theorem prover to formally prove the correctness of our translation function. We started the formal proof when we discovered inconsistencies in the handling of null references in an old version of the translation function. Fixing these inconsistencies led to the development of the notion of well-formed predicates (chapter 5.2), and we started a simple version of the correctness proof using Isabelle/HOL.

Later, we discovered additional correctness problems in our translation function regarding the range of quantifiers, in particular how the JML `\old` keyword and the choice of variable types affects the set of objects being quantified over. As our initial correctness proof did not model any types or heap states at all, this issue went undetected for some time. After we discovered these additional issues, we significantly extended the formal correctness proof, adding an explicit representation of the Alloy and JML syntax trees and modeling the language semantics using evaluation functions.

In this chapter, we will first explain the decisions and definitions used for formally modeling Alloy, JML and our translation function. We will then describe the structure of our correctness proof.

In particular, the proof about the translation function will show:

- The output JML expression compiles; it is well-typed.
- The output JML expression does not throw exceptions or produce undefined results.
- The output JML expression evaluates to true if and only if the input Alloy formula was valid.

7.1. Type System Formalization

Because the Java classes are used as the source for defining the Alloy signatures, we first formalize the necessary portions of the Java type system. This will be used by both the Alloy and JML formalizations.

```
typedecl JClassDecl — Java class declaration
typedecl JField     — Java field declaration
typedecl Object     — Object instance (at runtime)
typedecl VariableName

datatype JType = JBoolean | JInt | JNullType | JClass JClassDecl
```

These are the basic declarations for the formalized type system. The data type *JType* represents a Java type reference. For simplicity, we left out interface types, generics, and the primitive types other than `boolean` and `int`. Also, we added *JNullType*, which is the type of the null literal. The null type will allow us to assign a static type to every JML expression. In addition to the definition above, we will use *JObject* to refer to the class type of `java.lang.Object`.

On these types, we inductively define the subtype relation $T \sqsubseteq S$:

```
inductive isSubtypeOf :: "JType ⇒ JType ⇒ bool" ("_ ⊑ _")
where
  type_reflexive[simp]:         "T ⊑ T"
| type_trans[trans]:            "⟦T ⊑ U; U ⊑ S⟧ ⟹ T ⊑ S"
| classes_are_objects[intro]:   "JClass C ⊑ JObject"
| classes_are_nullable[intro]:  "JNullType ⊑ JClass C"
| type_inheritance: "⟦Some C1 = parentClass C2⟧ ⟹ JClass C1 ⊑ JClass C2"
```

The null type is a subtype of every type except for the primitive types `boolean` and `int`.

For objects, we assume the existence of a function *objectType* that returns the type of the object instance (like `Object.getClass()`):

```
fun objectType :: "Object ⇒ JType"
```

The *objectType* is guaranteed to be a class-type, it cannot be a primitive type or the null type.

For fields, we assume the existence of two functions *declaringType* and *fieldValueType* that return the class that declares the field, and the static type of the field's value, respectively.

```
fun declaringType :: "JField ⇒ JType"
fun fieldValueType :: "JField ⇒ JType"
```

The *declaringType* is guaranteed to be a class-type, while the *fieldValueType* has no restrictions.

Finally, we define the data type *JVal* to model the possible values of a JML expression:

```
datatype JVal =  BoolVal bool | IntVal int | null | ObjRef Object
abbreviation "true ≡ BoolVal True"
abbreviation "false ≡ BoolVal False"
```

7.2. Alloy Formalization

Alloy is a small language with simple semantics. However, for the purposes of this proof, a full formalization of Alloy would still be a significant task. Instead, we simplify Alloy further: first, we will only concentrate on a subset of the Alloy language that we think is the most important for the correctness proof of our overall translation approach. We will not prove all of the translation rules in chapter 5 correct, but only those that correspond to the basic Alloy language constructs modeled here.

As a second simplification, we make some adjustments to the Alloy type system to bring it closer to that of Java/JML. An Alloy type represents the type of a relation. We will define an Alloy type *AType* to be the list of the relation's column types. To represent the column types, we will use the Java types defined in the previous section.

```
type_synonym AType = "JType list"
```

This direct usage of Java types within Alloy relation types helps make the translation proof significantly simpler. However, it does have some effects on the semantics of the modeled language, to the point where it differs from the real Alloy. The major consequence of using the Java type system is that no universal type `univ` exists. We will use the `Object` type as an approximation – it is a supertype of all types except for the primitive types.

One consequence of this approximation to the real Alloy semantics is that, in our model, Alloy expressions involving a union of primitive integers/booleans and other atoms cannot be well-typed. Another is that the built-in relations `univ` and `iden` are restricted to objects and do not contain primitive integers or booleans. This corresponds to the restrictions on the allowable input of our translation in chapter 3.4.

A related difference is that Java does not have the `none` type. Instead, we will approximate it using the type of the null literal, *JNullType*. This does not affect the semantics of the `none` relation as that is the empty set either way, but it causes some expressions that are well-typed in regular Alloy to be invalid in our formalization. This means our formal correctness proof does not extend to such expressions, though we believe our translation of `none` is correct nonetheless.

It should be noted that the Alloy language also has union types. Our formalization omits these completely. While our translation can handle union types by translating them to a common base type, in our formalization we expect the input expression to be already typed using that common base type.

We now define an Alloy *Atom* as an analogue to the Java value *JVal*, except that we do not allow null references:

```
datatype Atom = IntAtom int | BoolAtom bool | ObjAtom Object
type_synonym Tuple = "Atom list"
type_synonym Relation = "Tuple set"
```

Tuples are defined as list of atoms, and relations as sets of tuples. This definition allows relations with mixed arities (tuples of different lengths). Indeed, we will define the Alloy semantics in an arity-independent way that allows such relations. However, in any well-typed Alloy expression, each relation will have a single, statically known arity.

7.2.1. Alloy dynamic state

We will now define the context information necessary to evaluate an Alloy expression:

```
type_synonym Env = "AVariable ⇀ Atom"
```

An Alloy environment maps Alloy bound variables to atoms. Alloy variables are declared as a pair of variable name and type. A variable is identified by both type and name: two variables with the same name but different types do not collide with each other. In the syntax tree, the variable type will be encoded in all references to the variable. This definition helps us to assign static types to Alloy expressions without having to use type environments.

An environment is called well-formed if it maps every variable to an atom of a compatible type:

```
definition wellFormedEnv :: "Env ⇒ bool"
where "wellFormedEnv e
          ≡ ∀v atom. e v = Some atom ⟶ atomtype atom ⊑ avartype v"
```

The environment is a partial function: the absence of a value for a variable means that the variable does not exist in the environment. Modeling the environment as a total function from all possible variables would cause the definition of well-formed environments to be unsatisfiable: for variables of type `none`, no value of a compatible type can exist. To keep the evaluation function total, evaluation of a missing variable is defined to return an empty set.

```
type_synonym Model = "Signature ⇒ Relation"
```

An Alloy model maps Alloy signatures to relations. A model is well-formed if the relation returned for a signature is compatible to that signature's type:

```
definition wellFormedModel :: "Model ⇒ bool"
where "wellFormedModel m ≡ ∀S. m S inType (sigtype S)"
```

Because both the environment and the model are mostly passed around recursively through the evaluation function and are rarely used individually, we combine them into a single parameter of the pair type `ModelEnv`. We call the `ModelEnv` well-formed if both its model and its environment are well-formed:

```
definition wellFormed :: "ModelEnv ⇒ bool"
where "wellFormed m ≡ wellFormedModel (model m) ∧ wellFormedEnv (env m)"
```

7.2.2. Alloy syntax and semantics

We now define the Alloy syntax tree using the mutually recursive type declarations `AExpr` and `BExpr`. `AExpr` refers to relational expressions; while `BExpr` refers to Alloy formulas. To syntactically distinguish Alloy operators from Isabelle/HOL operators, we mark them with the "$_A$" suffix. The evaluation functions are denoted as "`A⟦a⟧m`" and "`B⟦f⟧m`" respectively.

```
primrec evalA :: "AExpr ⇒ ModelEnv ⇒ Relation" ("A⟦_⟧_")
    and evalB :: "BExpr ⇒ ModelEnv ⇒ bool" ("B⟦_⟧_")
where
  "A⟦ALit i⟧m = {[IntAtom i]}"
| "A⟦AVar v⟧m = (case (env m v) of Some atom ⇒ {[atom]} | None ⇒ {})"
| "A⟦ASig S⟧m = model m S"
| ...
```

Literals evaluate to a singleton set. Variables are read from the environment; and signatures are read from the model.

```
| "A⟦a +_A b⟧m = A⟦a⟧m ∪ A⟦b⟧m"
| "A⟦a ._A b⟧m = relational_join (A⟦a⟧m) (A⟦b⟧m)"
```

Union are directly mapped to the corresponding operators for HOL sets. The relational join is implemented using a seperate definition:

```
definition relational_join :: "Relation ⇒ Relation ⇒ Relation"
where
 "relational_join r1 r2 = {t1@t2 |t1 t2 atom. t1@[atom]∈r1 ∧ [atom]@t2∈r2}"
```

An explanation of the Isabelle/HOL syntax involved here: the relational join is defined to be the set of tuples that are constructed by concatenating two tuples `t1` and `t2`, where the tuple `t1` concatenated with some atom is from the first relation, and the same atom concatenated with `t2` is an element of the second relation.

Next, we define the semantics of Alloy set comprehensions. While, Alloy allows an arbitrary number of variables in set comprehensions, we will only allow a single variable in set comprehensions. In addition, we introduce the construct of a "relation extension comprehension", using the custom syntax "`{v : a ‡ b}`":

```
| "A⟦{v : a | f}_A⟧m = {[atom] |atom. [atom]∈A⟦a⟧m ∧ B⟦f⟧m⟨v:atom⟩}"
| "A⟦{v : a ‡ b}_A⟧m = {[atom]@ts |atom ts. [atom]∈A⟦a⟧m ∧ ts∈A⟦b⟧m⟨v:atom⟩}"
```

The semantics of "`{v : a ‡ b}`" are: the expression *b* gets evaluated with the variable *v* set to each atom from the set `a`. Every tuple returned by the evaluation of *b* gets prefixed with the value of *v* that was used for its evaluation. This construct allows us to desugar set comprehensions involving multiple variables:
`{v1 : a1, v2 : a2 | f}` is equivalent to `{v1 : a1 ‡ {v2 : a2 | f}}`
This allows us to translate (and then proof) set comprehensions one variable at a time.

Finally, these are the semantics of Alloy formulas. The operators for Alloy formulas are marked with the "$_B$" suffix.

```
| "B⟦not_B f⟧m = (¬B⟦f⟧m)"
| "B⟦some_B a⟧m = (A⟦a⟧m ≠ {})
```

We left out quantified formulas because they can be desugared into quantified expressions with set comprehensions[1].

7.3. JML

Our JML formalization is roughly based on the Diploma thesis of Daniel Bruns [Bru09]. Following the approach in the Diploma thesis, we will model an evaluation function and a well-definition predicate for JML. Additionally, we will use a static well-typedness predicate. But we will also simplify JML for our formalization: we only define those language constructs that are used by the output of the translation function. JML is only "weakly side-effect free". That is, expressions in specifications cannot modify any existing objects on the heap, but they may allocate new objects. Since our translation does instantiate any objects, our subset of JML is complete side-effect free. We do not model side-effects in our formal JML semantics at all, which greatly simplifies the semantics.

As with our Alloy formalization, we identify variables by their signature (type, name, and nullability). Again, this helps us to assign static types to all JML expressions while avoiding the complexity of type environments.

Our JML syntax allows the use of an arbitrary JVal as a literal – apart from integer literals, boolean literals and the null reference, this also includes object literals. However, any expression containing such object literals is considered not to be well-typed. Since the correctness proof of the translation entails proving that the translated expression is well-typed, we can be sure that the output of the translation function is valid JML and does not contain any object literals. Object literals will get some limited use in the correctness proof of the translation.

7.3.1. JML dynamic state

```
record Heap =
    created :: "Object ⇒ bool"
    read    :: "Object ⇒ JField ⇒ JVal"
```

The heap structure captures the state of the Java heap at a single point of time. It provides the function '`created`' for determining whether an object is created in that heap state; and the function '`read`' for reading the value of a field. The read function is total, and assigns field values even to objects that are not yet created or are of a different class than the field's declaring type.

A heap is called *well-formed* if the following condition holds: if an object *obj* is created in the heap and its type is a subtype of the declaring type of a field *F*, then reading *obj.F* will result a value that is compatible with the field's type. Additionally, if such a read produces

[1]In the case of the `all` quantifier, we also need negation.

Definition	Type	Meaning
$S \sqsubseteq T$	`JType ⇒ JType ⇒ bool`	S is subtype of T or equal
`JT e`	`JExpr ⇒ JType`	static type of expression
`WT e`	`JExpr ⇒ bool`	is expression well-typed?
$free_J$ `e`	`JExpr ⇒ JVariable set`	set of free variables in JML expr
`s⟨old⟩`	`JState ⇒ JState`	Updates `s` to set the `inOld` flag
`s⟨var:=val⟩`	`JState ⇒ JVariable ⇒ JVal ⇒ JState`	Updates `s` to change local variable
`quantset s v`	`JState ⇒ JVariable ⇒ JVal set`	Quantifier range
`J⟦e⟧s`	`JExpr ⇒ JState ⇒ JVal`	JML evaluation function
`D⟦e⟧s`	`JExpr ⇒ JState ⇒ bool`	JML well-definedness predicate

Table 7.1.: Overview of important definitions for JML semantics

an object reference, the referenced object is also created in that heap. This means a heap cannot contain references to future objects. All other reads are left completely undefined.

```
record JState =
  locals :: "JVariable ⇀ JVal"    — Map of local variables to their values
  pre    :: Heap    — The pre-state heap
  post   :: Heap    — The post-state heap
  inOld  :: bool    — Whether evaluation is within \old(...)
```

The JState structure captures the context necessary to evaluate a JML expression. The mapping of local variables to values is a partial function.

A JState is called *well-formed* if all of the following conditions hold:

- Both its heaps are well-formed.
- Any objects that are created in the pre-state, must also be created in the post-state.
- The values stored in local variables are compatible with the variable types.
- Only variables marked as `nullable` may hold a null reference.

7.3.2. JML semantics

For quantifiers, an important consideration is the range of objects being quantified over. In JML, these are normally all objects of the bound variable's type, including those from future heap states. However, in KeY's JML*, quantifiers only range over the set of objects in the current heap state. In our formalization, we will use the JML* definition. We define the `quantset` to formalize the range of a quantifier:

```
definition quantset :: "JState ⇒ JVariable ⇒ JVal set"
where "quantset s var = {val. (valtype val ⊑ vartype var)
                              ∧ isCreatedOrPrimitive s val
                              ∧ (varnullable var ∨ val ≠ null)}"
```

The helper function `isCreatedOrPrimitive` returns whether an object is created in the current heap state. For all other values (ints, booleans, and null references), it always returns `true`.

The function `J⟦e⟧s` evaluates the JML expression e in the JState s:

```
primrec JEval :: "JExpr ⇒ JState ⇒ JVal" ("J⟦_⟧_")
where
  "J⟦JLit val⟧s = val"
```

```
| "J⟦JVar var⟧s = (case locals s var of Some val ⇒ val)"
| "J⟦a &&_J b⟧s = BoolVal (J⟦a⟧s = true ∧ J⟦b⟧s = true)"
| "J⟦a ==_J b⟧s = BoolVal (J⟦a⟧s = J⟦b⟧s)"
| "J⟦!_J e⟧s = BoolVal (J⟦e⟧s ≠ true)"
| "J⟦e instanceof_J T⟧s = BoolVal (valtype (J⟦e⟧s) ⊑ T ∧ J⟦e⟧s ≠ null)"
| "J⟦exists_J v; e⟧s = BoolVal (∃val ∈ (quantset s v). J⟦e⟧s⟨v:=val⟩ = true)"
| "J⟦e ._J F⟧s = (case J⟦e⟧s of ObjRef obj => read (heap s) obj F)"
| "J⟦old_J(e)⟧s = J⟦e⟧s⟨old⟩"
| "J⟦fresh_J(e)⟧s = BoolVal (case J⟦e⟧s of
                ObjRef obj ⇒ created (heap s) obj ∧ ¬created (pre s) obj
              | null       ⇒ False)"
```

If the expression e is not well-typed, the evaluation function may produce an undefined or misleading result. Examples are negating a non-boolean value, or reading a field from an object of an incompatible type. The JEval function may also produce an undefined value if an exception occurs during the evaluation. For example, a null reference may occur as the target for reading a field, leading to a an undefined result as the case distinction in the `J⟦e.F⟧s` semantics does not handle null references.

To check whether the evaluation of a JML expression is free of such errors, we use the defined-ness predicate `D⟦e⟧s`:

```
primrec JDefined :: "JExpr ⇒ JState ⇒ bool" ("D⟦_⟧_")
where "D⟦JLit v⟧s = True"
    | "D⟦JVar v⟧s = (locals s v ≠ None)"
    | "D⟦a &&_J b⟧s = (D⟦a⟧s ∧ (J⟦a⟧s ≠ true ∨ D⟦b⟧s))"
    | "D⟦a ==_J b⟧s = (D⟦a⟧s ∧ D⟦b⟧s)"
    | "D⟦!_J e⟧s = D⟦e⟧s"
    | "D⟦e instanceof_J T⟧s = D⟦e⟧s"
    | "D⟦exists_J v; e⟧s = (∀val ∈ (quantset s v). D⟦e⟧s⟨v:=val⟩)"
    | "D⟦e ._J F⟧s = (D⟦e⟧s ∧ J⟦e⟧s ≠ null ∧ isCreatedOrPrimitive s (J⟦e⟧s))"
    | "D⟦old_J(e)⟧s = D⟦e⟧s⟨old⟩"
    | "D⟦fresh_J(e)⟧s = (D⟦e⟧s ∧ ¬inOld s)"
```

The definition of `D⟦a &&_J b⟧s` implements the short-circuiting behavior of the Java logic operators. The other JML logic operators such as `a || b` or `a ==> b` are defined as syntax sugar for `a && b` with the appropriate negations.

Of particular importance is the defined-ness of the member-access operator '`e.F`': it requires '`e`' to be both non-null, and to refer to an object that exists in the current heap state[2]. Thus, the quantifier `(\exists List o; \old(o.next) == null)` is undefined if there are any newly created List objects in the post-state. A correct translation of the Alloy formula '`no List.next`' has to use the `\old`() operator outside of the quantifier:
`(\old(\exists List o; o.next == null))`

We then prove that the semantics of a JML expression do not depend on the value of local variables that are not free in the expression:

```
lemma eval_in_update[simp]: "var ∉ free_J e ⟹  J⟦e⟧s⟨var:=val⟩ = J⟦e⟧s"

lemma defined_in_update[simp]: "var ∉ free_J e ⟹ D⟦e⟧s⟨var:=val⟩ = D⟦e⟧s"
```

The two lemmas are proven by induction over the JML syntax tree. They are used in the correctness proof of the translation, as it is necessary for translation predicates to maintain their semantics while additional variables unrelated to the predicate are introduced.

[2]The requirement that the object be of a type compatible with the field's declaring type is not included in the definition of `D⟦e⟧s`. It instead follows from the well-typedness `WT e`.

7.4. Formalization of the translation function

The translation function is formalized as a pair of mutually recursive functions:

```
fun translateA :: "AExpr ⇒ Pred list ⇒ TContext ⇒ JExpr" ("ℰ«_‖_»_")
and translateB :: "BExpr ⇒ TContext ⇒ JExpr" ("ℬ«_»_")
```

The translation often creates quantifiers of the form `(\exists T v; p(v) && ...)`.
As defined previously in this thesis, we would need to use a "unique fresh variable name" to ensure that the bound variables of multiple generated quantifiers do not collide. However, formalizing these would require the translation function to maintain global mutable state.

Instead, we will allow re-use of variable names, and avoid collisions by allocating variables names in a top-down manner: The translation context maintains a set of variables that the translated code might refer to and thus must not be hidden by quantifiers. When a quantifier is generated, it picks a variable name that is not in the set, and performs any nested calls to the translation function using a modified translation context that includes the new variable in the used variable set. We will use the syntax "*c⟨usevar v⟩*" to denote the translation context based on c, except that the variable v was added to the set of used variables.

```
— Produces fresh variable name that does not collide with any name in the set
definition freshVariableName :: "VariableName set ⇒ VariableName"
where "freshVariableName s = (SOME n. n ∉ s)"

— Creates a fresh variable of the specified type and nullability
definition freshVariable :: "TContext ⇒ JType ⇒ bool ⇒ JVariable"
where "freshVariable c T nullable
        ≡ ⦇varname=freshVariableName (varname`usedvars c),
           vartype=T,
           varnullable=nullable⦈"

— Creates a JML exists quantifier using a new non-nullable variable of type T
definition makeExists
     :: "AColType ⇒ TContext ⇒ (JVariable ⇒ TContext ⇒ JExpr) ⇒ JExpr"
where "makeExists T c B ≡ (let v = freshVariable c T False in
                             (exists_J v; B v (c⟨usevar v⟩)))"
```

The translation context also maintains a map from Alloy variables to JML variables. Initially, this map is empty. When translating an Alloy construct that declares a new variable[3], this map will get extended by mapping the Alloy variable to the corresponding JML variable. To denote the this change of the translation context, we will use the syntax "$c\langle v_A \to v_J\rangle$".

Translation predicates could be modeled as functions "*JExpr ⇒ JExpr*". However, instead of using such functions bare, we wrap them in the distinct type "*Pred*", with an explicit apply operation:

```
datatype Pred = NewPred "JExpr ⇒ JExpr"

primrec applyPred :: "Pred ⇒ JExpr ⇒ JExpr" (infix "apply" 300)
where "(NewPred p) apply e = p e"
```

This allows the theorem prover to distinguish predicate application from other function applications of type *JExpr ⇒ JExpr*. This way, we can define introduction lemmas for predicate application as follows:

```
lemma wellFormedPred_D[intro]:
      "wellFormedPred c p ⟹ compat c s m ⟹ D⟦e⟧s ⟹ D⟦p apply e⟧s"
```

[3]In our desugared Alloy version, only the set comprehension declares new Alloy variables.

Without the explicit "*apply*" operation, this lemma would confuse the automatic proof search, as it would be usable in too many other contexts where it is not helpful.

We define the *nonnull* and *lift* predicates as explained in chapter 5.2:

```
definition nonnull :: Pred
where "nonnull ≡ NewPred (λe. e !=_J null)"

definition lift :: "JExpr ⇒ Pred"
where "lift objExpr ≡ NewPred (λe. e ==_J objExpr)"
```

Using these constructs, we can now define the translation function itself:

```
fun translateA :: "AExpr ⇒ Pred list ⇒ TContext ⇒ JExpr" ("ℰ«_‖_»_")
and translateB :: "BExpr ⇒ TContext ⇒ JExpr" ("ℬ«_»_")
where
  "ℰ«ALit i‖[p]»c = p apply (IntVal i)"
| "ℰ«AVar v_A‖[p]»c = p apply (case varmap c v_A of Some v_J ⇒ v_J)"
| "ℰ«ASig (TypeSig T)‖[p]»c = makeExists T c (λv c. p apply (JVar v))"
| "ℰ«ASig (PreFieldSig F)‖[p1,p2]»c
     = old_J(makeExists (declaringType F) c
              (λv c. p1 apply (JVar v) &&_J p2 apply (v._J F)))"
| "ℰ«ASig (PostFieldSig F)‖[p1,p2]»c
     = makeExists (declaringType F) c
         (λv c. p1 apply (JVar v) &&_J p2 apply (v._J F))"
| "ℰ«ASig FreshSig‖[p]»c
     = makeExists JObject c (λv c. p apply (JVar v) &&_J fresh_J(v))"
| — Union
  "ℰ«a +_A b‖ps»c = (ℰ«a‖ps»c ||_J ℰ«b‖ps»c)"
| — Relational Join
  "ℰ«a ._A b‖ps»c
     = makeExists (hd (AT b)) c
         (λv c. ℰ«a‖(take (arity a - 1) ps)@[lift (JVar v)]»c
            &&_J ℰ«b‖lift v#drop (arity a - 1) ps»c)"
| — Set comprehension
  "ℰ«{v_A : a | f}_A‖[p]»c
     = makeExists (avartype v_A) c
         (λv_J c. p apply (JVar v_J) &&_J ℰ«a‖[lift v_J]»c &&_J ℬ«f»c⟨v_A→v_J⟩)"
| — Relation extension comprehension
  "ℰ«{v_A : a ‡ b}_A‖p#ps»c
     = makeExists (avartype v_A) c
         (λv_J c. p apply (JVar v_J) &&_J ℰ«a‖[lift v_J]»c &&_J ℰ«b‖ps»c⟨v_A→v_J⟩)"
| "ℬ«not_B b»c = !_J ℬ«b»c"
| "ℬ«some_B a»c = ℰ«a‖replicate (arity a) nonnull»c"
```

7.5. Concepts for the proof

We already have a notion of well-formed JML heaps, well-formed states and well-formed Alloy models. We will now also define what "well-formed" means for translation contexts and translation predicates.

A translation context c, is called well-formed if its set of used variables is finite, and all JML variables in the image of the Alloy→JML variable map are contained in the set of used variables. The first constraint is used to ensure that the infinite set of possible variable names is never exhausted. The second constraint is necessary so that newly created fresh variables do not collide with those used in the variable map.

The well-formedness of translation predicates in more involved. It is dependent on a translation context c:

- If the predicate is applied to a well-typed expression, the resulting expression is well-typed and of type `boolean`.
- If applying the predicate to an expression e introduces additional free variables in the resulting expression, those free variables must be in the set of used variables in the translation context c.

Additionally, for every well-formed JState s, these conditions hold:

- Applying the predicate to a well-defined expression produces an expression that is also well-defined.
- Applying the predicate to an expression acts like variable substitution: if two expressions are semantically equivalent, the resulting expressions after applying a predicate are also equivalent.
- Applying the predicate to the null literal results in an expression that evaluates to `false`.

Next, we define a function *state2model* that takes a JML state *s* and returns the corresponding Alloy model:

```
primrec state2model :: "JState ⇒ Model"
where
 "state2model s FreshSig = {[ObjAtom obj]
     |obj. ¬created (pre s) obj ∧ created (post s) obj}"
| "state2model s (TypeSig T) = {[atom]
     |atom. atomtype atom ⊑ T ∧ isCreatedOrPrimitive s (atomval atom)}"
| "state2model s (PreFieldSig F) = {[ObjAtom obj, atom]
     |obj atom. created (pre s) obj ∧ objectType obj ⊑ declaringType F
              ∧ read (pre s) obj F = atomval atom}"
| "state2model s (PostFieldSig F) = {[ObjAtom obj, atom]
     |obj atom. created (post s) obj ∧ objectType obj ⊑ declaringType F

              ∧ read (post s) obj F = atomval atom}"
```

We now define the relation "*compat*" that specifies compatibility between a *JState* and a *Model* in a given translation context:
The *JState s* is compatible with the *ModelEnv m* in the translation context *c* if all of the following conditions hold:

- *s* is a well-formed state.
- *c* is a well-formed translation context.
- The model of *m* is equal to *state2model(s)*.
- For every entry $(v_J \mapsto v_A)$ in the translation context's variable map, the JML variable v_J has the same value in the state *s* as the Alloy variable v_A in the environment of *m*. Also, if the value is an object, it must be created in the post-state.
- All variables marked as used in the translation context are defined in the state *s*.
- Finally, the state *s* is not marked as being inside the `\old()` operator.

The last point is a critical precondition of the translation function: within the `\old()` operator, the post-state is completely unavailable and any references to field signatures in the post-state could not be translated. Also, quantifiers would be limited to objects in the pre-state. The condition that any object referred to in the environment must be created exists for a similar reason: if the environment contained an object that is not created in any of the two heap states, that object would not be included in any of the quantifiers we are generating.

Finally, we define the concept of a translation predicate accepting an Alloy atom:

```
abbreviation predAccepts :: "JState ⇒ Pred ⇒ Atom ⇒ bool" ("_ ~ _ ~ _")
where "s~p~a ≡ (J⟦p apply (JLit (atomval a))⟧s = true)"
```

To test whether a predicate accepts an Alloy atom, we create a JML literal with the atom's value, apply that literal to the predicate, and evaluate the resulting expression. This definition depends on a *JState* because the predicate might access local variables. Note that if the atom is an object, this definition ends up using object literals. These do not really exist in JML, and are not well-typed in our formalization. However, they are sufficient for the purpose of the *predAccepts* definition.

This definition can be extended to a list of predicates accepting a tuple. For this, we will use the syntax "*s~ps~~t*". A predicate list accepts a tuple iff the predicate list has the same length as the tuple, and every predicate accepts its respective atom.

7.6. The translation proof

The proof operates by induction over the syntactic structure of the expression being translated.

For an Alloy relation expression `a`, the induction hypothesis is:

$compat\ c\ s\ m$	Compatibility between JState s and ModelEnv m
$\wedge\ wellFormedPreds\ c\ ps$	The predicates are well-formed.
$\wedge\ length\ ps = arity\ a$	The number of predicates matches the relation's arity.
$\wedge\ WTA\ a$	The Alloy expression is statically well-typed.
$\wedge\ free_A\ a \subseteq dom\ (varmap\ c)$	All free Alloy variables are mapped to JML variables.
$\Longrightarrow$	
$D⟦\mathcal{E}«a\|ps»c⟧s$	The translated expression does not throw exceptions.
$\wedge\ (J⟦\mathcal{E}«a\|ps»c⟧s = true$	The translated expression evaluates to true
$\longleftrightarrow (\exists t \in A⟦a⟧m.\ s\text{\~{}}ps\text{\~{}\~{}}t))$	iff there is a tuple that satisfies the predicates.

For an Alloy formula *f*, the induction hypothesis is:

$compat\ c\ s\ m$	Compatibility between JState s and ModelEnv m
$\wedge\ WTB\ f$	The Alloy formula is statically well-typed.
$\wedge\ free_B\ f \subseteq dom\ (varmap\ c)$	All free Alloy variables are mapped to JML variables.
$\Longrightarrow$	
$D⟦\mathcal{B}«f»c⟧s$	The translated expression does not throw exceptions.
$\wedge\ (J⟦\mathcal{B}«f»c⟧s = true$	The translated expression evaluates to true
$\longleftrightarrow B⟦f⟧m)$	iff the Alloy formula is valid.

The induction hypothesis can only be used if a compatible *ModelEnv* exists for the recursive call. However, if a quantifier ends up quantifying over an empty set (for example, "`all x : none | F`"), no such *ModelEnv* can exist. In this case, the induction hypothesis isn't necessary to show the well-definedness and semantic equality of the subexpression, as the JML existential quantifier will never evaluate its subexpression. However, we still need to show the static well-typedness of the quantifier body. Because we cannot satisfy the "*compat c s m*" premise of the induction hypothesis, well-typedness cannot be proven as part of the induction hypothesis described above. Therefore, we prove well-typedness in a separate induction over the translation function, using very similar premises in the induction hypothesis, but just requiring a well-formed translation context instead of the full "compat" premise:

$\quad$ *wellFormedContext c* $\wedge$ *wellFormedPreds c ps* $\wedge$ *length ps = arity a*
$\wedge$ *WTA a* $\wedge$ *free*$_A$ *a* $\subseteq$ *dom (varmap c)*
$\Longrightarrow$

WT ($\mathcal{E}$«a‖*ps*»*c*)	The translated expression is well-typed.
$\wedge$ *JT* ($\mathcal{E}$«a‖*ps*»*c*) $\sqsubseteq$ *JBoolean*	The translated expression is of type `boolean`.
$\wedge$ *free*$_J$ ($\mathcal{E}$«a‖*ps*»*c*) $\subseteq$ *usedvars c*	All free variables in the output are marked as used in the translation context.

For Alloy formulas and the $\mathcal{B}$ translation function, a similar induction hypothesis is used. This translation well-typedness proof involves some manual instantiations and several intermediate steps that help the automatic proof search along, but did not pose any noteworthy difficulties.

As part of the translation semantics proof, we need to show that the generated JML* quantifiers have a range that includes all atoms that are potentially contained in the corresponding Alloy relation. For example, consider the set comprehension `{v : a | f}`, which is translated to `(\exists` $\mathcal{T}_v$[v]`;` p(v) `&&` $\mathcal{E}$«a‖*lift*(v)»$_c$ `&&` $\mathcal{B}$«f»$_{c'}$`)`. We need to show that any atom in the relation *A*⟦*{v : a | f}*⟧*m* is of a type that is a subtype of the quantifier's variable type $\mathcal{T}_1$[v]. This requires Alloy's type safety: because we require the Alloy input expression to be well-typed and the model m to be well-formed, Alloy's type safety provides us with the guarantee that the relation *A*⟦*{v : a | f}*⟧*m* is compatible with the static type of the set comprehension expression, which is defined to be the variable type. Additionally, we need to show that any atom in the Alloy relation is created in the current heap state, so that it is included in the JML* quantifier range. For this, we use the "*compat c s m*" premise: it implies that the current heap state is the post-state. We then show that any Alloy evaluation using a model that is compatible to some JState will produce a relation where all atoms are created in that relation's post-state. This is proven by a separate induction over the evaluation of Alloy expressions.

Finally, we show that the premises of our induction hypothesis are satisfied using the initial translation context when translating an Alloy formula that does not involve free variables:

definition *initialContext :: TContext*
where *"initialContext ≡ (| varmap=empty, usedvars={} |)"*

theorem
$\quad$**assumes** *"free*$_B$ *f = {}" "WTB f" "wellFormedState s" "¬inOld s"*
$\quad$**shows** *"let e = ($\mathcal{B}$«f»initialContext) in*
$\qquad\qquad$*WT e* $\wedge$ *D*⟦*e*⟧*s* $\wedge$ *(B*⟦*f*⟧*(state2modelenv s)* $\longleftrightarrow$ *J*⟦*e*⟧*s = true)"*

This concludes the formal correctness proof of the translation. The full Isabelle code can be found in the file `TranslationProof.thy`.

The formal proof presented here is only defined for some core Alloy language constructs. Many operators are missing; most critically, the proof did not include the transitive closure operator. We have another correctness proof (`OldProof.thy`) that includes more Alloy operators including the transitive closure; but that proof does not use the full JML semantics presented here. In particular, the Java type system and well-definedness of JML expressions are not modeled.

For the simplification step after the translation, we have verified the simplification rules in KeY by using JML specifications like the following:

```
/*@ public normal_behavior
  ensures
    (\exists SomeClass obj; f0() && x == obj && f1(obj))
   <==>
    (f0() && x instanceof SomeClass && f1((SomeClass)x));
  @*/
  public void existsSimp(/*@ nullable @*/ SomeClass x) { }
```

Here, `f0()` and `f1()` are pure methods that are otherwise unspecified. They are used as uninterpreted function symbols to stand as placeholders for the arbitrary JML expressions in the simplification rule.

8. Translated specifications in KeY

Using two example programs, we will show how to prove programs with specifications translated from Alloy are proven with KeY. We explore the limitations of KeY's automatic proof search in combination with the output from our translation, in particular translation of the transitive closure using query methods.

For the first example, we present a specification of a singly-linked list, which makes use of the transitive closure over the "`next`" field, as well as the `pow[]` function for explicit numbers of steps.

As a second example, we use the specification of the `add()` method in a binary search tree. This involves the more complicated transitive closure "`*(left+right)`", which leads to much more complex use of quantification.

We used an experimental KeY version that has improved support for recursively specified query methods. Unfortunately, a lot of time was consumed working around bugs in KeY. For example, if automatically generated query methods are placed at the end of the class instead of the beginning, KeY fails to parse the generated linked list example. We also encountered bugs that made it impossible to re-load a saved proofs, as well as unsoundness in the experimental "Replace query with postcondition" feature.

Unless mentioned otherwise, we will use the following settings for the KeY proof search strategy:

- Proof splitting: delayed
- Cut pruning: on
- Loop treatment: invariant
- Method treatment: contract
- Dependency contracts: off
- Query treatment: restricted
- Expand local queries: off
- Replace query by post: off
- Arithmetic treatment: basic
- Quantifier treatment: No Splits with Progs
- Auto Induction: Off

8.1. Linked List

The following is the definition of the class `Entry`, which represents a node in the linked list:

```
class Data { .. }
class Entry {
  /*@ nullable @*/ Entry next;
  /*@ nullable @*/ final Data data;

  //$ ensures no this.next'
  //$ ensures this.data' = d
  public /*@ pure @*/ Entry(/*@ nullable @*/ Data d) {
    this.data = d;
  }
}
```

The class `LinkedList` itself refers to the first entry, the `head` of the list. The head node does not contain any data, and exists even for the empty list. This simplifies insertion/removal at the beginning of the list, as the `head` pointer never changes; only the `next` pointers.

```
class LinkedList {
  final Entry head;
  int length;
  ...
}
```

Apart from the `head` pointer, our linked list implementation also maintains an integer that contains the length of the list. Within the proof, the length is used to ensure that the linked list is finite, which is necessary to ensure the termination of methods that traverse the list. Note that instead of an actual program field, a JML ghost field could have been used for this purpose instead.

Next, we specify the invariants of the linked list:

```
  //$ invariant this.length >= 0
  //$ invariant all k : Int | 0 =< k and k =< this.length
  //$                         implies (some this.head.(pow[next, k]))
  //$ invariant no this.head.(pow[next, add[this.length, 1]])
```

The second invariant ensures that the linked list is sufficiently long: every index from `0` to `this.length` must refer to a valid entry. Entry number 0 is the head node itself, which doesn't carry any data and exists even for empty lists. The third invariant ensures that the linked list ends after the last entry. It also implies that linked lists cannot be cyclic.

When translated to JML, these invariants are:

```
  //@ invariant this.length >= 0;
  //@ invariant (\forall int k; 0 <= k && k <= this.length
                  ==> getNext(this.head, k) != null);
  //@ invariant getNext(this.head, this.length + 1) == null;
```

The `getNext` method is defined recursively as described in chapter 6:

```
  /*@ public normal_behavior
      ensures index < 0 ==> \result == null;
      ensures index == 0 ==> \result == head;
      ensures index > 0 ==> \result ==
        (head != null ? getNext(head.next, index-1) : null);
      ensures index > 0 ==> \result ==
        (getNext(head, index-1) != null ? getNext(head, index-1).next : null);
      accessible \infinite_union(Entry e; e.next);
      static model helper strictly_pure nullable
      Entry getNext(nullable Entry head, int index);
  @*/
```

8.1.1. Constructor

For our first proof, we start with the constructor of the class:

```
//$ ensures this.length’ = 0
public /*@ pure @*/ LinkedList() {
  head = new Entry(null);
}
```

The correctness proof needs to show that the constructor establishes the object invariants. The first invariant about the non-negative length is shown trivially, as is the method's postcondition.

For the second invariant, all that needs to be shown is that `getNext(head, 0) != null` in the post-state. In KeY, this is represented by the formula "`LinkedList::getNext(@postHeap, self_0, 0) = null`" in the antecedent[1]. In that formula, `self_0` refers to the newly created head object, while `@postHeap` is a manually created abbreviation[2] for the heap state in the post-state of the constructor.

Because this new head object cannot be null, the formula "`self_0 = null`" occurs in the succedent.

To prove this goal, we need to use the contract of the `getNext()` method, which guarantees that "`getNext(..., self_0, 0) = self_0`".

8.1.2. Query method expansion

To use such a query method contracts within KeY, the option "query treatment" needs to be enabled.

Queries can also be expanded manually by the user when using KeY for interactive proving. When a query is expanded, KeY adds an assumption to the antecedent that equates the query call with a newly introduced result variable. Additionally, KeY adds an assumption that after an execution of the query method, the method result is equal to the result of the query call.

For the "`LinkedList::getNext(@postHeap, self_0, 0)`" query under discussion, the addition to the antecedent looks as follows:

```
!{heap:=@postHeap || head:=self_0 || index_0:=0}
    \<{method-frame(source=getNext(Entry, int)@LinkedList): {Entry queryResult;
          queryResult=getNext(head,index_0);
        }
      }\> !queryResult = res_getNext
& LinkedList::getNext(@postHeap, self_0, 0) = res_getNext
```

The method call can then be replaced by the method's contract as usual. This has the effect of splitting the proof into multiple goals:

[1]KeY always eliminates negations by moving the formula to the opposite cedent.

[2]`@postHeap = store(store(heapAfter_Entry, l_2, head, self_0), l_2, <initialized>, TRUE)`, where `l_2` is the linked list and `heapAfter_Entry` is the heap-state after the call to the Entry constructor.

1. "Pre" goal to show that the precondition of the query method is satisfied. This is shown trivially for the query methods generated by our translation, as they do not have any preconditions.
2. "Exceptional Post" goal to handle the case where the query method throws exceptions. Again, this case is trivial for our query methods, because they are specified to never throw exceptions.
3. "Post" goal to handle the case where the query method completes normally. Here, the postconditions of the query method are available as assumptions.

Using the "query treatment" setting to automatically perform query expansion, the correctness proof for the `LinkedList` constructor completes after 1467 automatic proof steps. A big problem here is that KeY performs query expansion only after proof splitting (e.g. for disjunctions in the antecedent). This causes the proof steps involving the query expansion to be done multiple times, on each side of the split. In the trivial example of the `LinkedList` constructor, interactively expanding the query just before KeY performs the first split reduces the proof size to 753 steps.

Because expanding a query introduces this additional nesting layer to the proof tree, as well as a new variable for the holding result, the sequent can quickly become unreadable when lots of query expansions are involved. This is a problem when debugging proofs: it can be hard to tell whether a proof does not complete automatically because it is impossible (for example, due to a missing precondition); or just because KeY's automatic proof search failed to find the proof so that manual steps may be necessary.

KeY query expansion is a breadth-first search over the queries to be expanded. This lets the proof search avoid cycles – for example, a depth-first search starting with the expansion of "`getNext(head, length)`" would continue with "`getNext(head, length - 1)`", "`getNext(head, length - 2)`" ad infinitum.

However, in some proofs, multiple successive query expansions are necessary. For example, in the `add()` method below, we need to expand "`getNext(head, 2)`" to "`head.next.next`", which requires three query expansions. In the breadth-first-search, KeY prefers query methods in "older" formulas on the proof branch over formulas that were just added to the sequent[3]. If the goal involving "`getNext(head, 2)`" is newly added, and there are many other unrelated `getNext()` calls in the sequent (for example, the object invariants), then the breadth-first search will try all other query expansion paths of length 3 before finding the correct one.

We can do better by adding a user-defined taclets that aggressively expands the base case of the recursive query method call. The custom rule `getNext_0` finds any occurrences of "`getNext(..., node, 0)`" and replaces them with "`node`":

```
getNext_0 {
  \find (LinkedList::getNext(heap, node, 0))
  \sameUpdateLevel
  "getNext_0 use":
    \replacewith (node);
  "getNext_0 heap valid":
    \add (==> wellFormed(node));
  "getNext_0 object valid":
    \add (==> node = null | boolean::select(heap, node,
                             java.lang.Object::<created>) = TRUE)
  \heuristics(simplify_boolean)
};
```

[3]This behavior implements the queue for the search.

This rule splits the proof into three goals. The first goal performs the replacement and is used to continue the actual proof. The other two goals are used to prove the implicit preconditions[4] of the `getNext()` method: it requires a well-formed heap and for all input objects to be created in the heap state used to call the method. These two goals are usually proven automatically by KeY in less than 20 steps each.

This custom rule has a relatively high priority in KeY: it runs before any case-distinctions, much earlier than normal query expansion. As a result, the base case of the `getNext()` recursion is expanded before any more complex cases. This decreases the effective height of the search tree that needs to be visited by KeY query expansion, dramatically decreasing the number of steps taken by the breadth-first search.

A similar custom taclet can be used when the step count is the constant 1: we replace "`getNext(node, 1)`" with "`node == null ? null : node.next`". This removes another level from the search tree.

Moreover, the custom taclets have an additional benefit: they allow expanding the query without introducing a new variable to represent the query result. This means the proof sequent tends use "`some_node.next`" instead of an automatically generated name. This greatly increases the readability when debugging a proof.

8.1.3. Lemma methods

Another approach for dealing with query expansion in KeY are *lemma methods*. These are normal Java methods with an empty method body. Using their postcondition, they provide a lemma to their call-site.

For example, consider the following method:

```
/*@ normal_behavior
  requires index >= 1 && index <= length + 1;
  requires (\forall int k; 0 <= k && k <= length ==> getNext(head, k) != null);
  ensures getNext(head, index) == getNext(head, index - 1).next;
@*/
private /*@ helper strictly_pure @*/ void lemma_expandNext(int index) { }
```

The postcondition of this method is a simple tail-recursive expansion of the recursive `getNext` query. The preconditions are used to ensure that the objects involved are not null. The lemma method itself can be proven automatically using KeY.

To use the lemma, the programmer calls the lemma method like a normal Java method. When proving the call-site, this has the effect of splitting the proof into the "Post", "Exceptional Post" and "Pre" goals as usual for any method call. In the "Post" goal, the lemma is usable as an assumption, and instantiated for the parameters passed to the method. In the "Pre" goal, the preconditions for the lemma method need to be shown – which is trivial in the case of `lemma_expandNext`, as the index tends to already be known to be from a valid range, and the second precondition is a simple copy of one of the object invariants.

Effectively, `lemma_expandNext` allows the programmer to give explicit hints about query expansion to the KeY theorem prover.

Another lemma that is often useful is the fact that nested query method invocations correspond to an addition of the step counts:

```
/*@ public normal_behavior
  ensures (\forall int a; a >= 0; (\forall int b; b >= 0; (\forall Entry head;
      getNext(getNext(head, a), b) == getNext(head, a + b))));
```

[4]This implicit precondition exists for every method call in the KeY logic.

```
@*/
static /*@ helper strictly_pure @*/ void lemma_getNextAdd() {}
```

This lemma can be proven by induction over `b`.

Finally, we prove that our object invariants ensure that linked lists cannot be cyclic:

```
/*@ public normal_behavior
  requires 0 <= k && k < l && k <= length;
  ensures getNext(head, k) != getNext(head, l);
@*/
private /*@ strictly_pure @*/ void lemma_acyclic(int k, int l) { ... }
```

The proof works by contraction: we assume that the two different indices `k` and `l` refer to the same entry. Then, we use `lemma_getNextAdd` to follow that `l + (l - k)` must also refer to that entry instance, as we traverse the cycle once more. We can repeat this addition until we exceed the list's length, thus showing that the repeated entry must be `null`. This is a contradiction to the fact that the initial indices were valid and thus referring to a non-null entry.

The proof is encoded as a Java program calling a variant of `lemma_getNextAdd()` using arguments to provide an instantiation for explicit integer values. The repetition is implemented as a Java `while` loop. This allows the KeY theorem prover to automatically complete the proof of `lemma_acyclic`.

8.1.4. add() method

Next, we prove the correctness of a method that inserts a node at the beginning of the linked list:

```
//$ ensures this.head.^next'.data' = this.head.^next.data + d
//$ ensures this.length' = add[this.length, 1]
//$ modifies this.head.next, this.length
public void add(Data d) {
  lemma_acyclic();
  Entry newEntry = new Entry(d);
  newEntry.next = head.next;
  head.next = newEntry;
  length++;
}
```

The contract of this method is underspecified and allows the method to insert the new node in an arbitrary position.

Translated to JML, the contract looks as follows:

```
//@ ensures (\forall Data obj88;
//@       (\exists int num9; num9 > 0 && getNext(this.head, num9) != null &&
    getNext(this.head, num9).data == obj88)
//@  <==> (\exists int num10; num10 > 0 && \old(getNext(this.head, num10)) !=
    null && \old(getNext(this.head, num10).data) == obj88) || d == obj88);
//@ ensures this.length == \old(this.length) + 1;
//@ modifies this.head.next, this.length;
```

In this kind of specification involving the transitive closure and set operators, the Alloy specification is much more readable than the JML specification.

When trying to prove this method, we run into a problem: because we modify the `next` member, the `getNext` calls in the post-state may return different values than the same calls in the pre-state.

Thus, the first goal in our proof should be to establish a relation between the post-state `getNext` and the pre-state `getNext`:

```
(\forall int n; n>0 ==> getNext(this.head, n + 1) == \old(getNext(this.head, n)))
```

Because KeY currently does not support JML assertions, we added this formula as an additional postcondition. Alternatively, we could have interactively introduced it as a cut in the KeY solver. Unfortunately the technique of lemma methods is not applicable in this case, as we cannot pass two different heap states to the method.

The proof of this lemma proceeds via induction over `n`. We start this induction manually in KeY using rule `autoInduct_Lemma`. For the base case `n = 1`, we need to show that

```
getNext(this.head, 2) == \old(getNext(this.head, 1))).
```

Note that at this time, the antecedent still holds all our previous assumptions, including the validity of the object invariants in the pre-state. Unfortunately, the prover requires an enormous number of steps to automatically solve this goal, because KeY does not pick the correct order to expand the queries. For example, the third invariant "`getNext(head, length + 1) = null`" is expanded first.

Because the recursive specification comes with both head- and tail-recursive specifications, each recursive query expansion enables not one, but two further expansions. Additionally, every such useless expansion tends to introduce additional case distinctions due to the null check in the `getNext` specification. All together, there is a very quick exponential blow-up in the proof size, which already makes the case `n = 1` extremely slow. In our testing, the proof search aborted with an out-of-memory error after more than 500000 steps.

If the user-defined taclets discussed in section 8.1.2 are used, the proof for the base case of the induction completes dramatically faster: It is found automatically in 3851 steps with 174 branches.

For the step case of the induction, we need to expand the queries `getNext(@postheap, head, n + 1)` and `getNext(heap, head, n + 1)`. Using `lemma_acyclic`, the induction step can then be proven automatically.

After the relation between the pre- and post-state `getNext` method is established, the remaining postconditions of the `add()` method can be proven automatically.

8.1.5. getEntry() helper method

To access an entry by index in the Java program, we internally use the `getEntry` method:

```
//$ requires n >= 0
//$ ensures result = this.head.(pow[next, n])
private /*@ helper strictly_pure nullable @*/ Entry getEntry(int n)
{
  int i = 0;
  Entry e = head;

  /*@ loop_invariant 0<=i && i<=n && e == getNext(head, i);
    assignable \strictly_nothing;
    decreases n-i; @*/
  while (i < n) {
    if (e != null)
      e = e.next;
    i++;
  }
  return e;
}
```

The proof for this method completes automatically in 677 steps using KeY's automatic query expansion. Note that loop invariants are currently not support by Alloy2JML, so they need to be written in JML.

8.1.6. removeAt() method

We now consider a method that removes an element from the linked list:

```
//$ requires index >= 1 and index =< this.length
// All entries before 'index' will be unchanged
//$ ensures all n : Int | n >= 0 and n < index
                  implies this.head.(pow[next', n]) = this.head.(pow[next, n])
// All entries after 'index' will move to the previous position.
//$ ensures all n : Int | n >= index implies this.head.(pow[next', n])
                                      = this.head.(pow[next, add[n, 1]])
// Length will be reduced by one
//$ ensures this.length' = sub[this.length, 1]
//$ modifies this.length, this.head.(pow[next, sub[index,1]]).next
public void removeAt(int index) {
  lemma_acyclic();
  lemma_expandNext(index);
  Entry e = getEntry(index - 1);
  e.next = e.next.next;
  length--;
}
```

With both query expansion and auto-induction enabled, KeY will find the correctness proof for this method fully automatically in 4079 steps. Note that the automatic proof search fails without the explicit query expansion using `lemma_expandNext`.

8.2. Binary Search Tree

Our second example is an unbalanced binary search tree. Here, we use the transitive closure `this.*(left+right)` to refer to the set of all sub-nodes of the current node. The following is the definition of the class `Tree`:

```
final class Tree {
  /*@ nullable @*/ Tree left;
  /*@ nullable @*/ Tree right;

  int height;
  int value;

  //$ invariant all n : this.*(left + right) {
  //$   n.height >= 0
  //$   some n.left implies n.left.height < n.height
  //$   some n.right implies n.right.height < n.height
  //$ }
  ...
}
```

The `height` field along with its invariant is used to disallow cyclic or infinite trees. This is necessary to ensure that operations on the tree terminate. Translated to JML, the invariant is:

```
//@ invariant (\forall Tree n; (\exists int num; num >= 0 && query(this, num, n))
//@               ==>   n.height >= 0
//@                     && (n.left != null ==> n.left.height < n.height)
//@                     && (n.right != null ==> n.right.height < n.height)
//@           );
```

The method `query` is a model method generated by Alloy2JML. It is the same as the `hasNode` example query in chapter 6.2:

```
/*@
  public normal_behavior
  ensures steps < 0 ==> \result == false;
  ensures steps == 0 ==> (\result <==> root == node && root != null);
  ensures steps > 0
    ==> (\result <==> root != null && (   query(root.left,  steps - 1, node)
                                       || query(root.right, steps - 1, node)));
  ensures steps > 0 ==> (\result <==> node != null &&
      (\exists Tree parent; query(root, steps - 1, parent)
                         && (parent.left == node || parent.right == node)));
  accessible \infinite_union(Tree obj; obj.left),
             \infinite_union(Tree obj; obj.right);
  static model helper strictly_pure
  boolean query(nullable Tree root, int steps, nullable Tree node);
@*/
```

So far, our invariant does not ensure that the binary tree is sorted. We use an Alloy predicate to formalize this condition:

```
  /*$ pred Tree.isSorted {
        all d : this.*(left+right) {
            all dl : d.left.*(left+right)  | dl.value < d.value
            all dr : d.right.*(left+right) | dr.value > d.value
          }
      }
  */
```

Alloy2JML converts this predicate into a model method:

```
/*@ public normal_behavior
    ensures \result <==>
      (\forall Tree d; self != null
                    && (\exists int num3; num3 >= 0 && query(self, num3, d))
        ==>   (\forall Tree dl; dl.left != null && (\exists int num1; num1 >= 0
                                                  && query(d.left, num1, dl))
                              ==> dl.value < d.value)
          && (\forall Tree dr; d.right != null && (\exists int num2; num2 >= 0
                                                  && query(d.right, num2, dr))
                              ==> dr.value > d.value));
    static model helper strictly_pure boolean isSorted(nullable Tree self);
@*/
```

Now, consider the constructor that creates a new tree with a single value:

```
  //$ ensures this.isSorted'
  public /*@ pure @*/ Tree(int initialValue) {
    this.value = initialValue;
  }
```

For this constructor, KeY automatically finds the proof in 11242 steps[5]. When using custom taclets for the base cases of the `query` function (null argument or `steps=0`), the same proof takes 4348 steps.

[5]cut-pruning enabled; query treatment=restricted; auto-induction disabled

8.2.1. contains() method

We now consider a simple recursive method for searching a value in the binary tree:

```
//$ requires isSorted[this]
//$ ensures result = True iff v in this.*(left+right).value
//$ measured_by this.height
/*@ strictly_pure @*/ boolean contains(int v) {
  if (v < this.value)
    return left != null && left.contains(v);
  if (v > this.value)
    return right != null && right.contains(v);
  return true;
}
```

Unfortunately, KeY is unable to find the proof for this method automatically. The proof search already stops in the simple case where "`v == this.value`". In this case, all that needs to be shown is that "`v in this*.(left+right).value`", or in KeY's logic:

```
\exists Tree obj48;
     (  (obj48.<created> = TRUE | obj48 = null)
      & !obj48 = null
      & (\exists int num4; (num4 >= 0 & query(self, num4, obj48) = TRUE)
                           & obj48.value = v))
```

The problem is that KeY does not find the instantiations `obj48=self` and `num4=0`. After explicitly providing these instantiations, the proof for the case "`v == this.value`" can be completed automatically. In fact, reordering the quantifiers and providing the instantiation `num4=0` is sufficient; KeY then is able to find the instantiation for `obj48` automatically.

Similar issues occur in other parts of the proof. For example, at one point, the automatic KeY proof search performs the cut "`self.right.height >= 0`". To show that this expression is true, all that is necessary is that the object invariant is instantiated for "`n = self.right`" and "`num = 1`". Again, KeY fails to perform this instantiation automatically and requires user intervention.

We experimented with user-defined taclets that automatically introduce assumptions of the form "`query(node, 0, node) = TRUE`" for any node object occurring in the sequent ("`node.<created> = TRUE`" assumption). Additionally, for every assumption of the form "`query(node1.right, steps, node2) = TRUE`", we introduce an additional assumption "`query(node1, 1 + steps, node2) = TRUE`".

Because this taclet introduces assumptions about query calls for "`self`" and "`self.right`", it helps KeY to automatically find the instantiations above. However, the taclet also causes new problems: it causes the proof to run into a loop, creating infinitely many assumptions in the series "`query(self, 1, self.right)`", "`query(self, 2, self.right.right)`" etc.

This happens because the KeY uses the new assumptions to instantiate the quantifier in the object invariant. This creates a formula with an additional level of "`.right`". KeY then performs a cut to to test whether this new object is null. In the proof branch where it isn't, our taclets then create `query` calls for the object expression with the additional `.right`. This enables further instantiations of the quantifier, thus sending the prover into a loop.

For proofs about the binary search tree, we had success with an approach that uses the KeY proof search on minimalist settings, combined with manual proof steps and the use of an SMT solver.

We use KeY with the following settings:

- Proof splitting **off**
- Query treatment **off**
- Quantifier treatment **none**
- User-defined taclets **enabled**, including the `add_assumption` taclets.

In this mode, KeY performs few automatic steps. For the most part, it just performs symbolic execution of the Java program, eliminating the modal operators. As query and quantifier treatment is disabled, the `add_assumption` taclets cannot cause loops in the proof.

For the most part, the resulting sequent stays readable after the automatic proof search stops[6]. For any incomplete branch of the proof, we would then try to see if the goal can be solved automatically. We first the automatic KeY proof search using the options at the beginning of this chapter, turning on proof splitting, query and quantifier treatment, while turning off the `add_assumption` taclets. If this search is not successful, we revert the proof state.

Alternatively, we let KeY transform the problem into first-order logic and pass it to the Z3 SMT solver[DMB08]. If Z3 finds a fully automatic proof, we can use it in KeY to close the current goal. Z3 can handle complex quantifiers better than KeY. However, Z3 still has problems with "`hasNode`"-style query methods. Even if Z3 is passed the contract for these methods (which KeY does not do automatically), it usually fails to find non-trivial proofs involving query expansion.

If neither KeY nor Z3 can find an automatic proof for the current goal, we manually perform the next proof steps. These usually involve providing explicit instantiations, or manually triggering query expansion.

In the case of the `contains()` method, the initial automatic steps leave the proof with 7 open goals. After expanding the "`isSorted(self)`" query on all of these branches[7], three of the goals can be solved automatically by Z3. The automatically closed goals correspond to the non-recursive code paths through the `contains()` method: returning true if the value is found, returning false if `left` is null, and returning false if `right` is null.

The remaining four goals deal with the recursive `contains()` calls: First, we need to show that the preconditions of the recursive call met. That is, if the current node (`this`) of the tree is sorted and the invariants are valid, then the same holds for the `this.right` child node. Additionally, the recursion must be shown not to be infinite.

After manually splitting the conjunction of these three parts into separate goals, we can re-run the automatic proof search with the `add_assumption taclets`. This has the effect of skolemizing the quantifications in the `isSorted` definition and the invariant of the for the child node. With some automatic simplifications, the assumption "`query(self.right, num_2, n_2) = TRUE`" appears in the antecedent. This triggers our custom `add_assumption` taclet, adding the assumption "`query(self, 1 + num_2, n_2) = TRUE`". After this, the automatic proof search stops. At this point, Z3 can automatically solve all three cases, completing the proof for the precondition of the recursive call.

The next open goal involves showing the method's postcondition ("`result = True iff v in this.*(left+right).value`") holds, under the assumption that the postcondition already holds for recursive call. We proceed by case distinction on the value of `result`. The case

[6]if quantifier treatment was enabled, normalization of quantifier bodies seriously harms the readability
[7]This can be done automatically using the 'replace query by postcondition' setting

`result=false` can be automatically solved by Z3. In the `result=true` case, we need to re-run the KeY proof search. Again, this skolemizes the quantifier introduced by the translation of the transitive closure in the postcondition, thus allowing the `add_assumption` taclet to introduce an assumption of the form "`query(self, 1 + num_2, n_2) = TRUE`". After this step, automatic proof search stops, and the remaining goal can be solved automatically by Z3.

The remaining two goals again deal with the pre- and postconditions of the recursive call, this time on the left child. The proof proceeds analogously to the right child.

Note that without the `add_assumption` taclet, we would have needed to provide several instantiations for quantifiers manually, as neither KeY nor Z3 try to instantiate quantifiers with `1 + num_2` unless this expression already occurs somewhere in the sequent.

Instead of using the user-defined `add_assumption` taclet, we can also use an equivalent lemma method:

```
/*@
    ensures (\forall Tree root, entry; (\forall int depth; query(root.left,
        depth, entry) ==> query(root, depth+1, entry)));
    ensures (\forall Tree root, entry; (\forall int depth; query(root.right,
        depth, entry) ==> query(root, depth+1, entry)));
@*/
/*@ helper strictly_pure @*/ static void lemmas_for_Z3() {}
```

The lemma method has the advantage that the lemma is passed to Z3, whereas user-defined taclets are invisible to the SMT solver. Using this lemma method, the `contains()` method can be proven with minimal interactivity: use KeY proof search on minimal settings to eliminate the modal operators; then pass all remaining goals to Z3 – they all complete automatically.

Note that when using such lemmas, the same caveats apply as with the `add_assumption` taclet: with quantifier treatment enabled, the KeY proof search tends to run into loops. Unlike user-defined taclets, they cannot be easily toggled on/off in the KeY interactive prover[8]. It would be nice if KeY allowed the selection of user-defined taclets for translation to SMT, as it already does for built-in taclets.

8.2.2. contains_loop

Let us consider an alternative implementation of the `contains` method using a loop instead of recursion:

```
//$ requires isSorted[this]
//$ ensures result = True iff v in this.*(left+right).value
/*@ strictly_pure @*/ boolean contains_loop(int v) {
  lemmas_for_Z3();
  Tree n = this;
  int steps = 0;
  /*@ loop_invariant steps >= 0 && query(this, steps, n)
          && (\forall Tree obj; (\exists int num4; num4 >= 0; query(this, num4,
               obj)) && obj.value == v
                              ==> (\exists int num5; num5 >= 0; query(n, num5
                                  , obj)));
    modifies \strictly_nothing;
    decreases n.height;
  @*/
  while (true) {
```

[8]It is possible to do so using the `hide` and `insert_hidden` rules.

```
      Tree child;
      if (v < n.value) {
        child = n.left;
      } else if (v > n.value) {
        child = n.right;
      } else {
        return true;
      }
      if (child == null)
        return false;
      steps++;
      n = child;
    }
  }
```

The loop invariant deals with both the path from `this` to the node `n`, and the path from `n` to whichever node contains the value we are looking for. Consider what happens in a step of the loop, as we change the variable `n` to point to one of its children: the path from `this` to `n` gets extended by one step at its tail; while the path from `n` to `obj` gets reduced by one step at the head. This proof was the primary motivation for why we need both head- and tail-recursive definitions for the `query` method.

The previously shown `add_assumption` lemma only works for extending paths at the head. For `contains_loop`, we also add equivalent tail-recursive lemmas:

```
(\forall Tree root, entry; (\forall int depth; query(root, depth, entry) &&
    entry.left != null ==> query(root, depth+1, entry.left)));
(\forall Tree root, entry; (\forall int depth; query(root, depth, entry) &&
    entry.right != null ==> query(root, depth+1, entry.right)));
```

The initial proof attempt with KeY leaves us with 10 open proof goals. All of these can be solved automatically by Z3, although two of them required a long computation time of more than one minute[9].

8.2.3. add() method

Finally, we will consider the `add()` method that adds a value to the tree. We will implement this by traversing the tree in the same way as the `contains()` method. If the value is found to be already present, it cannot be added again as the `isSorted` definition prohibits duplicate values. If this recursion ends without finding the value, a new node is created and added as a child to the last node visited by the recursion.

The step of actually adding the new node is performed by the `addLeftChild` helper method:

```
//$ requires no this.left
//$ requires v < this.value
//$ requires isSorted[this]
//$ ensures all steps : Int, a : Tree - fresh | a.(pow[left+right, steps]) = a
    .(pow[left'+right', steps]) - fresh
//$ ensures this.left' in fresh and this.left'.value' = v and no this.left'.(
    left' + right')
//$ ensures this.isSorted'
//$ modifies this.left, this.height
private void addLeftChild(int v) {
  left = new Tree();
  left.value = v;
  if (height == 0)
```

[9]The default timeout for SMT invocations in KeY is 5 seconds.

```
    height = 1;
  }
```

As with any method that changes the structure of the tree, it is important to have a postcondition that relates the query method in the post-state with the pre-state. For `addLeftChild`, the first postcondition has this purpose. It specifies that the results of the query method are unchanged for any existing nodes, though newly added objects may be added to the tree. Translated to JML, the first postcondition looks as follows:

```
//@ ensures (\forall int steps; (\forall Tree a; !\fresh(a) ==> (\forall Tree
    obj80;
  !\fresh(obj80) && \old(query(a, steps, obj80)) <==> query(a, steps, obj80) &&
     !\fresh(obj80))));
```

This postcondition is shown by induction over the number of steps. In the induction step, query expansion is necessary to reduce the `n+1` case to the case for `n` steps. The remaining proof goals complete automatically with Z3.

```
  //$ requires isSorted[this]
  //$ // returns false if the node already was present; true if it was added
  //$ ensures result = False iff v in this.*(left + right).value
  //$ // v is added to the set of values
  //$ ensures this.*(left' + right').value' = this.*(left + right).value + v
  //$ ensures this.isSorted'
  //$ // added nodes are newly created objects
  //$ ensures (this.*(left' + right') - this.*(left + right)) in fresh
  //$ modifies this.*(left+right).left, this.*(left+right).right, this.*(left+
      right).height
  //@ measured_by this.height;
  public boolean add(int v) {
    if (v < this.value) {
      if (left != null) {
        boolean result = left.add(v);
        updateHeight();
        return result;
      } else {
        addLeftChild(v);
        return true;
      }
    }
    if (v > this.value) {
      ...
    }
    return false;
  }
```

For brevity, the case `v > this.value` is omitted. Its implementation and proof are exactly analogous to the `v < this.value` case. The `updateHeight()` method is a helper that recomputes the `height` field after changes to the subnodes.

Here is the contract of the `add()` method translated to JML:

```
  //@ requires isSorted(this);
  //@ // returns false if the node already was present; true if it was added
  //@ ensures \result == false <==> \old((\exists Tree obj134; (\exists int num10
      ; num10 >= 0 && query(this, num10, obj134)) && obj134.value == v));
  //@ // v is added to the set of values
  //@ ensures (\forall int obj140; (\exists Tree obj141; (\exists int num11;
      num11 >= 0 && query(this, num11, obj141)) && obj141.value == obj140) <==> \
      old((\exists Tree obj147; (\exists int num12; num12 >= 0 && query(this,
      num12, obj147)) && obj147.value == obj140)) || v == obj140);
```

```
//@ ensures isSorted(this);
//@ // added nodes are newly created objects
//@ ensures (\forall Tree obj153; (\exists int num13; num13 >= 0 && query(this,
     num13, obj153)) && (\fresh(obj153) || (\forall int num14; num14 >= 0 ==>
    !\old(query(this, num14, obj153)))) ==> \fresh(obj153));
//@ // generated from modifies clause:
//@ ensures (\forall Tree obj164; !\fresh(obj164) && (\forall int num15; num15
    >= 0 ==> !\old(query(this, num15, obj164))) ==> obj164.left == \old(obj164.
    left));
//@ ensures (\forall Tree obj178; !\fresh(obj178) && (\forall int num16; num16
    >= 0 ==> !\old(query(this, num16, obj178))) ==> obj178.right == \old(obj178
    .right));
//@ ensures (\forall Tree obj192; !\fresh(obj192) && (\forall int num17; num17
    >= 0 ==> !\old(query(this, num17, obj192))) ==> obj192.height == \old(
    obj192.height));
//@ modifies \infinite_union(Tree obj; obj.left), \infinite_union(Tree obj; obj
    .right), \infinite_union(Tree obj; obj.height);
```

The `add()` proof itself is quite complex, requiring 75 interactive steps, around 31000 automatic steps in KeY, and numerous subgoals closed by Z3 invocations. We used a similar approach as with the previous methods: we run the KeY on minimalist settings with the `addAssumption` taclet; and use Z3 or continue the proof manually where the minimalist proof search stops. Most of the interactive steps are providing instantiations for quantifiers; a few steps are manually performing query expansion. After the `updateHeight()` call, we use the dependency contract for the query method (using its `accessible` clause) to show that the tree structure is not changed by a method call that modifies only the `height`.

In conclusion, the `hasNode`-style query methods used for the general case of the transitive closure are problematic for automatic proof search. Failures in automatic proof search require the user to understand the generated JML and how it relates to the original Alloy specification. This is a significant restriction on the usability of Alloy2JML. When looking at the causes of failing proofs, this was mostly because KeY failed to instantiate quantifiers. Especially the quantification over the number of steps caused trouble, as KeY did not try instantiations with "`steps+1`" unless that term already occurs in the sequent. Lemmas in the style of "`add_assumption`" would help with this, but send the automatic proof search into a loop. Fortunately, we can side-step this issue by delegating most of the proof search to the SMT solver Z3.

A different translation of transitive closures might have done better with the KeY proof search (e.g. using the `\reach` predicate). However, as the example of the linked list showed, our chosen approach works well in cases where the program also involves explicit step counts.

9. Conclusion

We developed the Alloy2JML tool that is capable of translating Alloy formulas to JML. Our tool uses a custom input format in which a Java program is annotated with specifications using Alloy formulas, similar to the TestEra and JForge tools. For the most part, the translation works by converting relational operators into first-order logic by quantifying over the elements of the relation. For the transitive closure, our translation introduces recursively specified model methods. We then used Alloy2JML to specify the behavior of two example programs, a linked list and an unbalanced binary search tree. We explored the automatic verification of the example programs against the output of our translation using the KeY theorem prover. We found techniques for automatically proving some methods; however the more complex methods still required substantial user interaction with KeY.

Using Isabelle/HOL, we proved that our translation is correct for a subset of the Alloy language.

In addition to the translation to JML, Alloy2JML provides an Alloy model as output. This allows exploration of the space of behaviors allowed by the specification using the Alloy Analyzer.

9.1. Related Work

JForge [DCJ06] allows the use of Alloy specifications in method contracts on Java programs, with syntax similar to ours. The tools transforms the Java program into a relational logic formula that relates the pre-state of a method to its post-state. The transformation unrolls loops for a bounded number of iterations. The resulting relational formula is then verified against the Alloy specification using Kodkod [TJ07], the relational engine used by the Alloy Analyzer. This will find any violations of the specification within the specified bounds on the heap size and number of loop iterations.

Our work complements JForge: while JForge only performs bounded verification, our tool can be used for a full correctness proof by translating the Alloy specification to JML, so that the program can be proven correct using a theorem prover like KeY. However, the minor differences in input syntax and the representation of Java types in Alloy (e.g. null atom vs. empty set) prevent the re-use of the same specification with both JForge and our translator.

JMLForge [DYJ08] allows bounded verification of JML contracts on Java programs. It uses the same approach as JForge, and translates both JML and the Java program into relational logic.

The Kelloy system presented in [Gei11] verifies Alloy models by translating them into KeY's first-order logic. The translation adds a relational theory to KeY by defining predicates for set membership and the various Alloy operators; and then defines rules for reasoning about these predicates by providing rules as KeY taclets. Kelloy does not use JML and is not specific to the verification of Java programs.

Our translation of the transitive closure is based on the specification of the linked list by Gladisch and Tyszberowicz [GT13]. This specification was designed to work with both the KeY prover and the JET runtime testing tool. The latter tool requires specifications to be executable. Unfortunately, our translation from Alloy generally does not produce executable specifications. In our specification of the linked list, the Alloy code is explicitly quantifying over the number of steps using the `pow` function. The resulting JML specifications here are executable with the exception of some clauses where we omitted an upper bound on the number of steps. However, as soon as the Alloy specification makes use of the transitive closure or other set operations (for example, in the `removeAll()` method), the output JML specification will make use of unbounded quantifiers, both over the number of steps in the transitive closure, and over objects involved in set operations. This means that the output of Alloy2JML is not suitable for JML tools like JET that depend on runtime assertion checking.

The OpenJML [Cok13] toolchain can parse and typecheck the output of our translation with minor modifications: the KeY-specific "`strictly_pure`" needs to be replaced with "`pure`", and the use of "`\infinite_union`" needs to be replaced with a static data group. The OpenJML runtime assertion compilers fails due to the reasons mentioned above. Extended static checking in OpenJML fails for outputs involving the transitive closure, as OpenJML currently does not support the use of recursively specified query methods.

9.2. Future Work

The Alloy2JML input format currently only supports a few contract clauses for lightweight method specifications. It is missing support for exceptional behavior specifications, and for multiple specification cases in general. A future version of the tool should support more of the JML constructs. Ideally, we would adopt the JForge Specification Language[Yes09], thus allowing the re-use of specifications written for JForge. Additionally, we should add support for loop invariants, so that these too can specified using the Alloy language.

Moreover, Alloy2JML could be extended by supporting some of the Alloy languages features that are currently restricted (chapter 3.4). In particular, the following language constructs were left unsupported due to time constraints:

- User-defined Alloy functions
- `disj` keyword
- Built-in functions like `max[]`

For the use of the KeY prover, we have found custom taclets to be useful to prioritize query expansion for the base cases of the recursively specified query methods. In the future, such custom taclets could be automatically generated by Alloy2JML.

Appendix

A. Example Translations

The following table contains a list of example Alloy formulas and the resulting JML after translation and simplification. In the examples, "`this`" refers to a non-nullable variable of type `Tree`.

Alloy	**JML**
`some this.left`	`this.left != null`
`some this.left.left`	`this.left != null && this.left.left != null`
`this.left in this.right`	`this.left != null ==> this.right == this.left`
`no (this.left & this.right)`	`this.left == null || this.right != this.left`
`result < this.left.value`	`\result < (this.left != null ? this.left.value : 0)`
`no this.left.value`	`this.left == null`
`this in {x : Tree | x.value > 0}`	`this.value > 0`
`this.(left+right) in {x : Tree | x.value > 0}`	`(\forall Tree obj; this.left == obj || this.right == obj ==> obj.value > 0)`
`#(this.left) = this.height`	`(this.left != null ? 1 : 0) == this.height`
`#(Tree & fresh) = 5`	`(\num_of Tree obj1; \fresh(obj1)) == 5`
`no this.(~left)`	`(\forall Tree obj1; obj1.left != this)`
`(x -> y) in iden`	`x == y`
`no (Root <: parent)`	`(\forall Root obj; obj.parent == null)`

`all o: FSObject - Root {`
`  some o.parent`
`}`
	`(\forall FSObject o; !(o instanceof Root) ==> o.parent != null)`

`head.(pow[succ, index]) = head.(pow[succ, sub[index, 1]]) .succ`	`getSucc(head, index) == (getSucc(head, index - 1) != null ? getSucc(head, index - 1).succ : null)`
`v in this.*(left+right).value`	`(\exists Tree obj; (\exists int num; num >= 0 && query(this, num, obj)) && obj.value == v)`

B. Implementation Notes

The Alloy2JML tool is implemented in Java. It uses OpenJML[Cok13] as a library for parsing the input Java source code into an abstract syntax tree. OpenJML is used instead of a simple Java parser so that JML annotations like `pure` and `helper` can be detected easily.

The Alloy annotations in the input file are discovered using a small hand-written lexer. Multiple successive `//$`-style comments are concatenated together. Using the Alloy Analyzer (version 4.2) as a library, we parse the annotations and run Alloy's semantic analysis on them in the context of a module generated from the Java classes. The translation functions $\mathcal{B}$ and $\mathcal{E}$ recursively traverse and translate the Alloy syntax tree using the visitor pattern. $\mathcal{B}$ is implemented in `FormulaConvertVisitor.java`; $\mathcal{E}$ is implemented in `RelationConvertVisitor.java`.

A custom JML syntax tree is used for the generated output expressions. Simplifications are performed as the output JML syntax tree is constructed.

We use JUnit for testing the translator. The tests can be found in `ConversionTests.java`.

The source code for Alloy2JML is provided on the CD as an Eclipse project. The referenced OpenJML and Alloy libraries are included. To run Alloy2JML, pass the file name of the Java input file on the command line:

```
cd Implementation
java -cp bin;lib/alloy4.2.jar;lib/openjml.jar
     alloy2jml.Main ../BinaryTree/Tree.java
```

The output file will be placed in the directory `"BinaryTree/out"`.

Bibliography

[BHS07] B. Beckert, R. Hähnle, and P. H. Schmitt, *Verification of object-oriented software: The KeY approach.* Berlin, Heidelberg: Springer-Verlag, 2007.

[Bru09] D. Bruns, "Formal Semantics for the Java Modeling Language," Diplomarbeit, Karlsruhe Institute of Technology, 2009.

[Cha07] P. Chalin, "A sound assertion semantics for the dependable systems evolution verifying compiler," in *In International Conference on Software Engineering*, 2007, pp. 23–33.

[Cok13] D. Cok, "OpenJML - formal methods tool for Java and the Java Modeling Language," 2013. [Online]. Available: http://openjml.org/

[Dar11] J. D. Darcy, "JEP 120: Repeating Annotations," 2011. [Online]. Available: http://openjdk.java.net/jeps/120

[DCJ06] G. Dennis, F. S.-H. Chang, and D. Jackson, "Modular verification of code with SAT," in *Proceedings of the 2006 international symposium on Software testing and analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 109–120.

[DGT13] F. Damiani, C. Gladisch, and S. Tyszberowicz, "Refinement-based testing of delta-oriented product lines," in *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ser. PPPJ '13. New York, NY, USA: ACM, 2013, pp. 135–140.

[DMB08] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[DYJ08] G. Dennis, K. Yessenov, and D. Jackson, "Bounded verification of voting software." in *VSTTE*, vol. 5295. Springer, 2008, pp. 130–145.

[Gei11] U. Geilmann, "Verifying Alloy Models Using KeY," Diplomarbeit, Karlsruhe Institute of Technology, 2011.

[GT13] C. Gladisch and S. Tyszberowicz, "Specifying a linked data structure in jml for formal verification and runtime checking," in *Brazilian Symposium on Formal Methods (SBMF)*, ser. LNCS, L. de Moura and J. Iyoda, Eds. Springer, 2013.

[Jac06] D. Jackson, *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[KYZ$^+$11] S. A. Khalek, G. Yang, L. Zhang, D. Marinov, and S. Khurshid, "TestEra: A tool for testing Java programs using alloy specifications." in *International Conference on Automated Software Engineering*, P. Alexander, C. S. Pasareanu, and J. G. Hosking, Eds. IEEE, 2011, pp. 608–611.

[LPC$^+$11] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl, "JML Reference Manual," 2011. [Online]. Available: http://www.jmlspecs.org

[TJ07] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *In Tools and Algorithms for Construction and Analysis of Systems (TACAS*. Wiley, 2007, pp. 632–647.

[Wei11] B. Weiß, "Deductive verification of object-oriented software: Dynamic frames, dynamic logic and predicate abstraction," Ph.D. dissertation, Karlsruhe Institute of Technology, 2011.

[Yes09] K. T. Yessenov, "A Lightweight Specification Language for Bounded Program Verification," Master's thesis, Massachusetts Institute of Technology, 2009.

[Zav12] P. Zave, "Using lightweight modeling to understand chord," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 2, pp. 49–57, Mar. 2012.