6-1-2013

Enhancing the Expressiveness of the CleanJava Language

Melisa Vela
University of Texas at El Paso, smvelaloya@miners.utep.edu

Yoonsik Cheon
University of Texas at El Paso, ycheon@utep.edu

Comments:
Technical Report: UTEP-CS-13-33

Enhancing the Expressiveness of the CleanJava Language

Melisa Vela and Yoonsik Cheon

TR #13-33
June 2013

Keywords: formal specification; functional programming; functional program verification; intended function; CleanJava.

1998 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — Correctness proofs, formal methods; D.3.2 [*Programming Languages*] Language Classifications — Applicative (functional) languages; D.3.3 [*Programming Languages*] Language Constructs and Features — Classes and objects, control structures, inheritance, polymorphism, functions; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, logics of programs, specification techniques.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

Enhancing the Expressiveness of the CleanJava Language

Melisa Vela and Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
smvelaloya@miners.utep.edu; ycheon@utep.edu

***Abstract*—The CleanJava language is a formal annotation language for Java to support Cleanroom-style functional program verification that views a program as a mathematical function from one program state to another. The CleanJava notation is based on the Java expression syntax with a few extensions, and thus its vocabulary is somewhat limited to that of Java. This often makes it difficult to specify the rich semantics of a Java program in a succinct and natural way that is easy to manipulate for formal correctness reasoning. In this paper we propose to make the CleanJava language more expressive by supporting user-defined mathematical functions that are introduced solely for the purpose of writing annotations. A user-defined function is written in a notation similar to those of modern functional programming languages like SML and Haskell and has properties such as polymorphism and type inference. We also explain how the notion of functions fits in the object-oriented world of Java with concepts like inheritance and method overriding. User-defined functions not only enrich the vocabulary of CleanJava but also allow one to tune the abstraction level of annotations. One contribution of our work is bringing the notion of functions as found in modern functional programming languages to an object-oriented programming language in the context of writing annotations, thus blending the benefits of two programming paradigms.**

Keywords: formal specification, functional programming, functional program verification, intended function, CleanJava.

I. Introduction

In Cleanroom-style functional program verification, a program is viewed as a mathematical function that maps one program state to another, and a correctness proof of a program is performed by comparing two mathematical functions, the function implemented by the program, called a *code function*, and its specification, called an *intended function* [1] [2] [3]. The CleanJava language is a formal notation for annotating a Java program with intended functions and proving its correctness [4] [5]. It is an add-on notation to the Java language in that annotations like intended functions are written using Java program elements such as variables, fields, and methods. The CleanJava notation is based on the Java expression syntax with a few CleanJava-specific extensions such as mathematical structures (e.g., sets and sequences) and specification-only methods (see Section II-A). CleanJava promotes the use of more formal program comments for rigorous or formal correctness reasoning of Java programs.

Functional programming languages such as SML [6] and Haskell [7] are based on and mimic mathematical functions to the greatest extent possible. A functional program is simply an expression written in terms of functions, and executing the program means evaluating the expression. They not only have a solid theoretical basis but also are closer to the user by providing a higher level of abstractions. One key feature of functional programming languages is *referential transparency*, meaning that the evaluation of a function always produces the same result given the same parameters. This is due to the side-effect-freeness of functions and greatly facilitates the correctness reasoning of a program. Functional languages also support other interesting features like a lambda notation and higher order functions.

Since a functional program verification technique views a program as a mathematical function, it is quite natural to consider adopting functional programming language notations for specifying the behavior of a program. In this paper we propose to enhance the CleanJava notation by extending it with concepts and notations available from modern functional programming languages. In particular, we propose notations for defining mathematical functions and manipulating mathematical structures like sets and sequences for the purpose of writing CleanJava annotations. The semantics of the new notations are defined semi-formally by translating them to the CleanJava standard notation. At the semantic level, for example, a function is interpreted as a query method available only for writing annotations. The proposed notations allow one to formulate problem or domain concepts in a more natural and abstract fashion, and the resulting annotations are not only more concise and understandable but also better manipulatable for formal correctness reasoning.

The remainder of this paper is structured as follows. In Section II below we provide a quick overview of CleanJava and functional programming by focusing on the features that will be used in this paper. In Sections III and IV we first state the problem of the CleanJava language in supporting a user-defined vocabulary for writing annotations and then describe main challenges in introducing a functional notation to CleanJava. In the next two sections, we propose and illustrate new notations for writing mathematical functions (Sections V) and manipulating mathematical structures like sets and sequences (Section VI). In Section VII we apply our proposed notation to an example, and in Section VIII we conclude this paper along with discussions of some aspects of our notations.

II. Background

A. The CleanJava Language

The CleanJava language is a formal annotation language for Java to support Cleanroom-style functional program ver-

```
1   //@ [n > 0 → result := crList(1,n)→select(int x; n % x == 0)]
2   public List<Integer> factors(int n) {
3     //@ [r, i := new ArrayList<Integer>(), 1]
4       List<Integer> r = new ArrayList<Integer>();
5       int i = 1;
6
7     /∗@ [r, i := appendAll(r, crList(i,n)→select(int x; n % x == 0)),
8       @    anything] @∗/
9     while (i <= n) {
10      //@ [n % i == 0 → r, i := append(r,i), i + 1 | true → i := i + 1]
11        //@ [n % i == 0 → r := append(r,i) | true → I]
12        if (n % i == 0) {
13          //@ [r := append(r,i)]
14          r.add(i); }
15        //@ [i := i + 1]
16        i++; }
17
18    //@ [result := r]
19    return r; }
20
21  /∗@ public List<Integer> crList(int i, int j) {
22    @   List<Integer> r = new ArrayList();
23    @   for (int k = i; k <= j; k++) { r.add(k); }
24    @   return r; } @∗/
25
26  /∗@ public List<Integer> append(List<Integer> l, int i) {
27    @   List<Integer> r = new ArrayList(l);
28    @   r.add(i);
29    @   return r; } @∗/
30
31  /∗@ public List<Integer> appendAll(List<Integer> l1,
32    @       List<Integer> l2) {
33    @   List<Integer> r = new ArrayList(l1);
34    @   r.addAll(l2);
35    @   return r; } @∗/
```

Fig. 1. Sample Java code annotated in CleanJava

ification [5]. In CleanJava, intended functions are written using *concurrent assignments* of the form $[x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n]$ stating that the new values of state variables x_i's in the output state are those of the expressions e_i's simultaneously evaluated in the input state. If a state variable doesn't appear in the left hand side, its value is assumed to remain the same. The expressions e_i's in the right hand side are written in the Java expression syntax with a few CleanJava-specific extensions.

Figure 1 shows a Java code snippet annotated in CleanJava. The code calculates all the factors of a given positive number, and each section of code is annotated with an intended function. In CleanJava, an annotation is written as a special comment, either preceded by //@ or enclosed in /*@ and @*/, and precedes the section of code that it annotates. For example, the annotation in line 1 is the intended function for the whole *factors* method, and line 3 annotates the code in lines 4-5; as shown, an indentation is used to associate an intended function with code that spans more than one line. The intended function in line 1 uses several CleanJava features. It specifies a partial function using an extended form of the current assignment notation called a *conditional concurrent assignment*. The condition preceding the arrow symbol (→) defines the domain of the function, thus specifying a partial function. The *factors* method is defined only for a positive value of *n*. As mentioned earlier, an intended function is written in the Java expression syntax with a few CleanJava-specific extensions. For example, the keyword *result* denotes the return value of a method, and the *select* operation is an "iterator" defined for a collection to select elements. This particular *select* expression selects all the elements of the collection *crList(1,n)* that can divide *n* evenly; note that an arrow notation (→) is used for an iterator to disambiguate it from a Java method invocation. The method *crList* defined in lines 21-24 is an example of a specification-only method, a method solely introduced for the purpose of writing intended functions. The example defines two more specification-only methods in lines 26-35, and all specification-only methods should be side-effect free. The annotations in lines 10-11 shows another feature of CleanJava. They specify different functions depending on conditions; each conditional concurrent assignment specifying a different function is separated by a vertical bar symbol (|), and the *I* symbol denotes an identity function that doesn't change the program state.

B. Functional Programming

Functional programming is a programming paradigm where function definitions are the basic components of programs. Modern functional programming languages like SML [6] and Haskell [7] are purely functional in that they contain no imperative constructs like variables, states and statements. They allow programmers to write concise code by providing language constructs and features of a higher level of abstractions, such as powerful type systems, type inference, recursive functions, high-order functions, lambda notations, and lazy evaluation. These make the functional programming paradigm a very powerful programming style. In Haskell, for example, the *factors* function can be defined as follows.

```
factors n = fromTo 1 n where
  fromTo y y = [y]
  fromTo x y = (if mod n x == 0 then [x] else []) ++ fromTo (x+1) y
```

The *where* clause introduces local definitions—e.g., a helper function *fromTo*–and [] and ++ denote a list and a list concatenation operation, respectively.

In most functional languages, lists are one of the most common ways to structure data, and in fact they are the central part of the language. In Haskell, for example, one can use a list comprehension notation to express the *factors* function succinctly as: factors n = [x|x ← [1..n], mod n x == 0]

Another key feature of functional programming languages is the elimination of side effects. There is no variable, no assignment statement, and thus no side effect. This feature makes it easier to reason about correctness of programs. It supports *referential transparency*, meaning that a function always returns the same result when applied to the same values. In reasoning, this means that a function application can always be replaced with its definition with an appropriate renaming of parameters, thus supporting the inference rule "substitution of equals for equals".

III. The Problem

The notation of CleanJava is based on the Java expression syntax in that CleanJava expressions are Java expressions with no side effect. This makes the CleanJava language more approachable to Java programmers. However, one downside of this approach is that the vocabulary for writing annotations is limited to that of Java. Besides, the Java vocabulary—e.g., program variables and methods—is implementation-oriented and thus is of algorithmic nature focusing on the "how" aspect of a program. Such a vocabulary is often not suitable for writing specifications. For writing intended functions, preferred is one focusing on the property or "what" aspect of a program, e.g., a term stating all the factors of a number or the property of one number being a factor of another, not the algorithm for or the calculation of factors.

The designers of the CleanJava language already recognized the need for problem or domain-specific vocabulary of a higher-level of abstractions and in fact provided CleanJava-specific extensions to the Java expression syntax, including a mathematical toolkit for sets and sequences and specification-only methods. As shown in the previous section, specification-only methods allow one to define and build one's own problem-specific vocabulary for writing intended functions without polluting the Java name space. To annotate the *factors* method, for example, we defined several new terms like *crList*, *append*, and *appendAll* that can be used only in annotations, not in Java code. They enrich the vocabulary for writing intended functions.

However, there are also shortcomings in using specification-only methods to build one's own vocabulary for writing intended functions. The shortcomings are mainly caused by the use of the Java method declaration syntax; remember that specification-only methods are Java methods declared inside annotations. The meanings of new terms like *crList* have to be defined algorithmically as a sequence of Java statements. The resulting annotations tend to be verbose, long and often less readable and understandable. In the code snippet of the previous section, for example, there are 22 lines of annotations out of 35 lines of source code, and thus 63% of source code are annotations. More than half of the annotation lines (59%) are for defining new terms, i.e., three specification-only methods. A more serious problem than the verboseness and understandability is that intended functions written using specification-only methods are less manipulatable for formal reasoning. This is again due to the algorithmic definition of a term. One of the key strengths of functional program verification is its support for equational reasoning by allowing "substitution of equals for equals". In reasoning using intended functions, for example, one can replace a function application with its definition. However, this is not possible for specification-only methods. For example, one cannot replace or expand the expression *crList(1,n)* with the *crList*'s definition because its definition is a Java code block, i.e., a sequence of Java statements.

An ideal notation would be one that allows one to build one's own vocabulary for writing intended function succinctly and that supports equational reasoning. One possibility would be to use mathematical functions defined in the style of pure functional programming languages like SML and Haskell. In fact, this idea has arisen at an early stage of the CleanJava language design [8] [5], and it has been used informally in an ad-hoc fashion when writing sample annotations or performing case studies [9] [10]. However, the idea still needs to be fully realized by considering its implications to the whole CleanJava language and by defining its precise syntax and semantics. Lack of a formal definition leads to not only confusions and inconsistency in using the notation but also lack of tool support; e.g., the current version of the CleanJava checker doesn't support a notation for user-defined mathematical functions yet [11].

IV. Challenges

There are several challenges in introducing a functional notation to the CleanJava language, and they are mainly due to the fact that it requires blending of two different programming paradigms. Some are simply syntactic or notational challenges while others are more of semantic nature. The notations of modern functional programming languages like Haskell and SML are different from that of Java. They provide more succinct and powerful notations for defining functions and writing expressions. For example, the type information of a function such as the argument types and the return type may be omitted from a function definition, as it is automatically inferred by the system, and a list can be constructed and expressed in a notation similar to the form of the mathematical set compression. In Java, however, a method signature has to be explicitly and completely specified, and there's no language construct similar to the set comprehension notation. On the semantic side, the main challenge is blending the concepts of functions and immutable values (e.g., lists) to the object-oriented conceptual framework of the Java programming language. For example, how does the notion of functions fit in the Java's conceptual framework of objects, classes, methods, inheritance, overriding, and dynamic binding, etc.? Are functions defined for objects and classes, and if so, can they be inherited by or overridden in subclasses?

As stated above the main challenge of our work is to make the notion of functions fit in the Java's object-oriented conceptual framework and notation. The key to our solution is to view a function as a specification-only method written using a Haskell-like functional notation. In other words, a function is sort of a syntactic sugar for defining a specification-only method in a more succinct, natural, and easy-to-manipulate way. To address the notational challenges, we tried to strike a balance between the succinctness and the Java-likeness of newly introduced notations. For this we make several parts of a function definition optional by either adopting defaults for or inferring the missing parts. In the following several sections we describe and illustrate our approach in detail by focusing on mathematical functions and a set comprehension-like notation.

$\langle CJStmt \rangle$ ::= $\langle$*fun-decl*$\rangle$ | ...
$\langle$*fun-decl*$\rangle$ ::= $\langle$*fun-header*$\rangle$ '=' $\langle CJExpr \rangle$
$\langle$*fun-header*$\rangle$ ::= '`fun`' [type] $\langle$*ident*$\rangle$ '(' [$\langle$*param-list*$\rangle$] '`)`'
$\langle$*param-list*$\rangle$ ::= $\langle$*param*$\rangle$ | $\langle$*param*$\rangle$ '`,`' $\langle$*param-list*$\rangle$
$\langle$*param*$\rangle$ ::= [$\langle$*type*$\rangle$] $\langle$*ident*$\rangle$

Fig. 2. Syntax for defining functions. An optional part is enclosed in square brackets.

V. Mathematical Functions

A. Basic Notation and Semantics

Figure 2 shows our proposed, basic syntax for defining mathematical functions in CleanJava. A function declaration denoted by a non-terminal $\langle$*fun-decl*$\rangle$ is introduced as a new kind of CleanJava statements; the non-terminal $\langle CJStmt \rangle$ represents all types of CleanJava statements [11]. A function declaration consists of a header and a body, and the body is simply a CleanJava expression (refer to [11] for CleanJava expressions). Note that the specification of the parameters and return type in the header is optional. Below is an example function declaration with the type information fully specified.

```
//@ fun int abs(int x) = x >= 0 ? x : −x
```

As hinted in the previous section, a function is interpreted as a specification-only method. This semantic interpretation has some nice consequences. There is no need to define a new syntax for applying functions, as the existing method invocation syntax (e.g., abs(-10)) will work. More importantly, a function declaration can be viewed as a syntactic sugar or a shorthand notation for defining a specification-only method. This means that there is no need to extend the Java's conceptual framework to support the notion of functions and associated features. As we will describe below, most features of functions can be nicely mapped to those of specification-only methods. For example, the above function can be desugared to the following specification-only method.

```
//@ public int abs(int x) { return x >= 0 ? x : −x; }
```

If a parameter type or the return type is omitted, it is inferred at the compilation time. For example, if omitted, the return type of the `abs` function can be inferred from the type of the parameter `x` and `x`'s use (e.g., `-x`) in the body expression.

A function declaration oftentimes defines a *polymorphic function,* a function that can be applied to values of different types. This is done by omitting the parameter types. If the type of the parameter `x` is omitted from the definition of `abs`, for example, the `abs` function can now be applied to different types of numbers such as Integer, Float, and Double. A polymorphic function is interpreted as a generic method (see below), and the most specific type (e.g., Number) will be inferred for a type parameter (e.g., T).

```
//@ fun abs(x) = x >= 0 ? x : −x

/*@ public <T extends Number> T abs(T x) {
  @   return x >= 0 ? x : −x; } @*/
```

B. Static Functions

A function is either *static* or *non-static*. By default, all functions are non-static in that they are interpreted as, or mapped to, non-static specification-only methods. A non-static function can refer to non-static members of the class, such as instance fields and methods. As in Java, one can also define a static function by using a modifier "static" as follow.

```
//@ static fun int abs(int x) = x >= 0 ? x : −x
```

A static function is of course mapped to a static specification-only method, and thus its definition cannot refer to non-static members such as instance fields and methods. A static function can be used only in a static context such as a static method and a static initialization block.

C. Inheritance of Functions

Since functions are members of classes, they can also have visibility and be inherited by subclasses. All functions are by default "public" functions. This is a slight deviation from Java in which fields and methods default to package-visible. However, we believe this is a good design because most functions are likely to be introduced for a public use. One can also define non-public functions by using modifiers like *protected*, *private*, and *package*; note that package is a new modifier that we propose specifically for CleanJava.

Since functions are mapped to specification-only methods, public or protected functions are inherited to subclasses, and subclasses can override inherited functions. If functions are overridden in subclasses, the functions to be applied are determined based on the dynamic type of the receiver; that is, functions are dynamically dispatched.

D. Abstract and Interface Functions

Just like specification-only methods, functions can be declared to be abstract by using the *abstract* modifier. An abstract function allows one to defer its definition to subclasses by omitting its body expression. It provides a way to write some high level or incomplete specifications that will be fleshed out by subclasses by leaving out certain details to be defined by subclasses.

A function can also be defined in an interface, for example, to specify the behavior of an interface method. This however causes a problem that Java avoided avidly. A function can be inherited multiply, e.g., one from the superclass chain and the other from the interface chain. This should be avoided because if it happens its meaning is undefined.

E. Local Functions

It is possible to introduce a locally-scoped function. For example, in addition to functions declared at the class member level, termed *member functions*, one can also introduce functions at the statement level, termed *statement functions*. A statement function is mainly for limiting the scope of a function to a single method body or a block of statements. It is also possible to define a function whose scope is a single CleanJava expression or statement (e.g., an intended function)

⟨*CJExpr*⟩ ::= ⟨*col-expr*⟩ | ...
⟨*col-expr*⟩ ::= [[`'new'`] ⟨*type*⟩] `'{'` ⟨*col-body*⟩ `'}'`
⟨*col-body*⟩ ::= ⟨*val-enum*⟩ | ⟨*val-seq*⟩
⟨*val-enum*⟩ ::= ⟨*CJExpr*⟩ | ⟨*CJExpr*⟩ `','` ⟨*val-enum*⟩
⟨*col-seq*⟩ ::= ⟨*CJExpr*⟩ [`','` ⟨*CJExpr*⟩] `'..'` ⟨*CJExpr*⟩

Fig. 3. Syntax for defining collection literals

using the "where" clause. The following code snippet shows three statement functions: *sum* in line 1 with the whole code block as its scope, *sumH* in line 2 with a single CleanJava expression as its scope, and *sumAll* in line 4 with a single CleanJava statement as its scope.

```
1   /*@ fun sum(a, 0) = sumH(a, 0, 0) where
2     @  fun sumH(a, i, s) = i >= a.length ? s : sumH(a, i+1, s+a[i]) @*/
3
4   //@ [r := sumAll(a)] where fun sumAll(a) = sum(a, 0)
5     //@ [r, i := 0, 0]
6     r = 0; int i = 0;
7
8     //@ [r, i := r + sum(a, i), anything]
9     while (i < a.length) {
10     //@ [r, i := r + a[i], i + 1]
11      r = r + a[i];
12      i++;
13    }
```

VI. Collections

Mathematical structures such as sets and sequences play an important role in specifying and verifying the behavior of Java program modules, as Java classes can be frequently modeled as sets or sequences. In fact, CleanJava already provides abstractions of these mathematical structures including sets, bags, sequences, and mappings [5]. These standard library classes define operations similar to those of Java collection classes. However, since these classes are intended for use in writing annotations, they are all immutable types; there is no method for changing the state of an object. For example, an *add* method creates and returns a new collection object instead of mutating the receiver object. In this section, we propose a new notation for creating and manipulating these mathematical structures. Our key approach is to map these mathematical structures to the standard CleanJava library classes by treating the new notation as a syntactic sugar and by desugaring it to the standard CleanJava notation.

A. Collection Literals

In CleanJava, there is no direct language construct for expressing collection literals. One has to represent a collection literal indirectly by defining a specification-only method (as done in Section II-A) or using a CleanJava standard library method such as fromArray, e.g., CJSet.fromArray(int[] {10, 20, 30}). We propose a new language construct to express a collection literal directly (see Figure 3), and below are shown some example collections literals specified in the new notation.

```
CJSet<Integer>{10, 20, 30, 40, 50}
{10, 20 .. 50}
{'a'..'z'}
```

⟨*col-body*⟩ ::= ⟨*val-comp*⟩ | ...
⟨*val-comp*⟩ ::= ⟨*CJExpr*⟩ `'|'` ⟨*var-decls*⟩ [`';'` ⟨*CJExpr*⟩]
⟨*var-decls*⟩ ::= ⟨*var-decl*⟩ | ⟨*var-decl*⟩ `','` ⟨*var-decls*⟩
⟨*var-decl*⟩ ::= [⟨*type*⟩] ⟨*ident*⟩ `':='` ⟨*CJExpr*⟩

Fig. 4. Collection comprehension notation

As shown, a collection literal can be expressed by either enumerating all its element values or specifying an interval for ordinal element values. An interval of values is specified by a pair of values, start and end values, separated by two dots (..); an optional step that defaults to 1 can also be specified after the start value. It denotes a collection of consecutive values starting at the start value and ending at the end value. The type of a collection literal is optional. If it's not specified, it defaults to CJSequence<T>, where T is the element type; CJSequence is a CleanJava standard library class representing an immutable sequence of values [5]. If specified, it should be one of the CleanJava standard collection classes or implement the java.util.Collection interface. The type of all the elements must be the same.

B. Collection Comprehension

A collection is often created based on existing collections. For this, we propose a *collection comprehension*, a construct similar to a mathematical set comprehension notation. For example, a collection $\{3x + 1 \,|\, x \in S \land x > 0\}$ is written in our proposed notation as:

```
{3*x + 1 | x := S; x > 0}
```

Figure 4 shows the complete syntax of our proposed collection comprehension notation. A collection comprehension consists of three parts: an expression, generators, and an optional predicate. The result expression (e.g., 3*x + 1) that produces the elements of the collection is typically written in terms of generators (e.g., x). Each generator is associated with a collection (e.g., x with S) that provides the values for the generator. The optional predicate filters the values of the generators. The result collection is obtained by evaluating the expression using generator values that satisfy the predicate.

The semantics of a collection comprehension can be defined more formally by translating a comprehension to a standard CleanJava expression using a collection iteration operation as follows.

$$\begin{array}{l} \{E \mid x_1 := C_1, x_2 := C_2, ..., x_n := C_n;\ B\} \equiv \\ C_1 \rightarrow \text{iterate}(T_1\ x_1,\ T_c\ r_1 = \{\}; \\ \quad r_1 = r_1 \cup C_1 \rightarrow \text{iterate}(T_2\ x_2,\ T_c\ r_2 = \{\}; \\ \quad\quad ... \\ \quad\quad\quad r_{n-1} = r_{n-1} \cup C_{n-1} \rightarrow \text{iterate}(T_n\ x_n,\ T_c\ r_n = \{\}; \\ \quad\quad\quad\quad r_n = r_n \cup (B\ ?\ \{E\} : \{\}))\ ...)) \end{array}$$

where T_i is the element type of C_i, T_c is the type of C_1, and the $\cup$ symbol denotes a collection operation adding all the elements of the second collection to the first. The *iterate* operation allows one to iterate over and manipulate the elements of a collection while accumulating the result (refer

```
1  //@ fun factors(m,n) = {x | x := {m..n}; n % x == 0}
2
3  //@ [n > 0 → result := factors(1,n).asList())
4  public List<Integer> factors(int n) {
5    //@ [r, i := new ArrayList<Integer>(), 1]
6      List<Integer> r = new ArrayList<Integer>();
7      int i = 1;
8
9    /*@ fun concat(l, s) = {x | x := l}.appendAll(s).asList()
10
11   //@ [r, i := concat(r, factors(i,n)), anything]
12   while (i <= n) {
13     //@ [n % i == 0 → r, i := concat(r,{i}), i + 1 | true → i := i + 1]
14       //@ [n % i == 0 → r := concat(r,{i}) | true → I]
15       if (n % i == 0) {
16         //@ [r := concat(r, {i})]
17         r.add(i); }
18       //@ [i := i + 1]
19       i++; }
20
21   //@ [result := r]
22   return r; }
```

Fig. 5. Revised sample Java code

to [5] and [11]). For example, the above comprehension is translated as follows, where T is the type of S.

```
{3*x + 1 | x := S; x > 0}
  ≡ S→iterate(int x, T r = T{};
                r = r.addAll(x > 0 ? T{3*x + 1} : T{}))
  ≡ S→collect{int x; x > 0; 3*x + 1}
```

As shown, if there is only one generator, the translation can be simplified by using the *collect* operation.

A collection comprehension can have more than one generator. The following comprehension, for example, has two generators (x and y) and produces a sequence consisting of sums of the pairs from two sequences, i.e., {11, 12, 21, 22, 31, 32}.

```
{x + y | x := {10, 20, 30}, y := {1, 2}}
```

As shown in this and previous examples, the type of the collection may be omitted, which defaults to that of the collection associated the first generator.

VII. Example

In this section we apply our proposed notation to the *factors* example in Section II-A. Figure 5 shows the example with intended functions written using the new notation. As shown, all the specification-only methods were got ridden of and instead introduced were two functions, a member function in line 1 and a statement function in line 9. The *factors* function in line 1 returns a sequence of numbers, consisting of factors of n that are larger than or equal to m. The *concat* function in line 9 concatenates a list and a sequence and returns the result as a list; the *asList* method is defined for the CleanJava collection library classes such as CJSequence. Both functions are defined using collection comprehensions. Several intended functions are now re-written using these two mathematical functions.

Are the new annotations better than the original ones? The annotations are now more concise and compact. They require a 55% less number of source code lines, occupying only 10 lines compared to 22 lines in the original example. Note that the number of the intended functions are the same, but the new annotations require only two lines for defining a new vocabulary (two mathematical functions) while the original require 13 lines for three specification-only methods. The new annotations are more abstract and understandable, as they are expressed using a vocabulary for the problem or domain concepts. For example, the *factors* function defines what factors are without concerning how they are calculated. The annotations are more manipulatable in formal reasoning. A function application can be replaced with its definition. In reasoning, for example, the term *factors(1,n)* appearing in the intended function in line 3 can be replaced with or expanded to $\{x \,|\, x := \{1..n\};\ n \,\%\, x == 0\}$.

VIII. Discussion

The example in the previous section also shows several aspects of our proposed notation that could be further improved. The notation requires an explicit conversion between different collections types, especially between CleanJava and Java collections. For example, the *factors* function in line 1 produces an instance of CJSequence<Integer>, and the result sequence has to be converted to List<Integer> in line 3 for an assignment to the *result* pseudo variable (of type List<Integer>). The other direction of conversion is shown by the *concat* function in line 9 that takes a list (l) and a sequence (s) and returns a concatenated list. The function is defined as $\{x \,|\, x := l\}.appendAll(s).asList()$, and the whole purpose of the comprehension subexpression is to convert the list to a sequence to invoke the *appendAll* method; note also that the sequence is eventually converted back to a list. The reason for these conversions is to manipulate collections in annotations by calling various library methods defined in the CleanJava collection classes. We are currently looking into this issue by considering several possible solutions, and an ideal solution would be to make all these conversions take place automatically. One possibility is to introduce a type coercion operator, say, $e : T$, where e is a collection expression and T is a collection type. Its meaning can be defined by translating it to a collection iterator as: $e{\rightarrow}iterate(T_e\ x, T\ r = \{\}; r = r.add(x))$. Another possibility is to have an implicit type conversion rule between Java collections and CleanJava collections. For a Java-to-CleanJava conversion we already have the CleanJava iterator notation, e.g., $l{\rightarrow}appendAll(s)$ meaning that l is first converted to a CleanJava collection and then the *appendAll* method is invoked. For a CleanJava-to-Java conversion we can use the context of a collection expression to perform an automatic conversion. If a CleanJava collection appears in a context where a Java collection is expected, we can automatically translate it to $e : T$, where T is a Java collection type required by the context.

Another improvement possible is to provide a set of collection manipulation operators. As in Haskell, for example, we may introduce an infix collection concatenation operator, say ++, to write an expression like l ++ s, which is short for $l{\rightarrow}appendAll(s)$. This may be a useful feature considering that collections like sets and sequences play an important role in writing model-oriented specifications. A good starting place would be the mathematical tool kit of other specification languages, e.g., Z specification language [12].

There are several other features of functional programming languages that we are currently investigating for an adoption in the CleanJava language, including higher order functions, lambda notations, and more concise function definition notations (e.g., pattern matching and splitting definitions). For example, a higher-order function allows one to write a reusable and flexible function by either taking functions as parameters or yields a function as its result, or both, and a lambda notation allows one to express a function on-the-fly in the spot where needed without defining it in a separate declaration statement. We believe the reusability and flexibility offered by these language constructs make the CleanJava language more expressible and usable. However, one concern is how to map these concepts to the Java's object-oriented conceptual framework. One possibility would be to use a *functional interface*—any interface that has exactly one explicitly declared abstract method—as done in the recent proposal of lambda expressions in Java 8 [13]. That is, a functional interface can be used for a method parameter when a lambda is to be supplied as the actual argument. Thus, a higher order function is simply a function whose parameter types or result type are functional interfaces. We may also adapt the lambda expression notation of Java 8 for expressing a lambda function in CleanJava.

There are two places that the design of our notation deviates from Java's language rules. One is the default visibility of a member function. In Java, when a class member such as a field or method has no modifier, it defaults to package-private. For a member function, however, we decided to make the default be *public*; for a package visible member function, one has to declare it explicitly by using a new *package* modifier. We believe this is a good design, as we think a member function is most likely to be introduced for a public use. However, this still needs to be validated, for example, through case studies. Another deviation is the possibility of multiple inheritance for member functions by allowing them to appear in interfaces. This is something we cannot avoid, as interfaces are an ideal place to write interface specifications, i.e., specifications for abstract methods declared in interfaces.

IX. Summary

In this paper we reported a preliminary design of our extended CleanJava notation for defining mathematical functions and manipulating mathematical structures like sets and sequences. Our motivation was to provide a more concise and richer notation for defining a problem or domain-specific vocabulary for writing CleanJava annotation by adopting concepts and constructs from modern functional programming languages such as SML and Haskell.

We are currently performing more realistic case studies to evaluate our proposed notation. As thus, we don't have a critical evaluation yet, but one nice thing about our approach is that we map the concept of functions nicely to existing concepts of CleanJava and Java. A function is essentially a specification-only method in CleanJava—and thus an observer method in Java—defined in a more succinct mathematical notation that is more understandable and more manipulatable as well in formal reasoning. The semantics of our notation is also defined by translating it to the standard CleanJava language.

Acknowledgment

This work was supported by NSF grant DUE-0837567.

References

[1] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore, *Cleanroom Software Engineering*. Addison Wesley, Feb. 1999.

[2] A. Stavely, *Toward Zero Defect Programming*. Addison-Wesley, 1999.

[3] Y. Cheon and M. Vela, "A tutorial on functional program verification," Department of Computer Science, The University of Texas at El Paso, Tech. Rep. 10-26, Sep. 2010.

[4] Y. Cheon, "Functional specification and verification of object-oriented programs," Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, Tech. Rep. 10-23, Aug. 2010.

[5] Y. Cheon, C. Yeep, and M. Vela, "The CleanJava language for functional program verification," *International Journal of Software Engineering*, vol. 5, no. 1, pp. 47–68, Jan. 2012.

[6] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML*. The MIT Press, 1997.

[7] G. Hutton, *Programming in Haskell*. Cambridge University Press, 2007.

[8] Y. Cheon, C. Yeep, and M. Vela, "CleanJava: A formal notation for functional program verification," in *ITNG 2011: 8th International Conference on Information Technology: New Generations, April 11-13, 2011, Las Vegas, NV*. IEEE Computer Society, 2011, pp. 221–226.

[9] C. Avila and Y. Cheon, "Functional verification of class invariants in CleanJava," in *Innovations and Advances in Computer, Information, Systems Sciences, and Engineering*, ser. Lecture Notes in Electrical Engineering, vol. 152. Springer-Verlag, Aug. 2012, pp. 1067–1076.

[10] Y. Cheon and C. Avila, "Constructing verifiably correct Java programs using OCL and CleanJava," Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, Tech. Rep. 13-15, Feb. 2013.

[11] C. Yeep and Y. Cheon, "CJC: an extensible checker for the CleanJava annotation language," Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, Tech. Rep. 13-29, May 2013.

[12] J. M. Spivey, *The Z Notation: A Reference Manual*, ser. International Series in Computer Science. New York, NY: Prentice-Hall, 1989.

[13] Oracle, "JSR 335: Lambda expressions for the Java programming language," 2012, date retrieved: May 16, 2013. Available from `http://jcp.org/en/jsr/detail?id=335`.