8-1-2010

Functional Specification and Verification of Object-Oriented Programs

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

Follow this and additional works at: http://digitalcommons.utep.edu/cs_techrep
Part of the Computer Engineering Commons
Comments:
Technical Report: UTEP-CS-10-23

Recommended Citation
http://digitalcommons.utep.edu/cs_techrep/23

This Article is brought to you for free and open access by the Department of Computer Science at DigitalCommons@UTEP. It has been accepted for inclusion in Departmental Technical Reports (CS) by an authorized administrator of DigitalCommons@UTEP. For more information, please contact lweber@utep.edu.
Functional Specification and Verification of Object-Oriented Programs

Yoonsik Cheon

TR #10-23
August 2010

Keywords: behavioral subtyping; correctness proof; functional verification; intended function; Cleanroom; Java

1998 CR Categories: D.2.4 [Software Engineering] Software/Program Verification — Correctness proofs, formal methods; D.3.2 [Programming Languages] Language Classifications — Object-oriented languages; D.3.3 [Programming Languages] Language Constructs and Features — Classes and objects; control structures; inheritance; polymorphism; 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.
Functional Specification and Verification of Object-Oriented Programs

Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, TX 79968
ycheon@utep.edu

Abstract—One weakness of Hoare-style verification techniques based on first-order predicate logic is that reasoning is backward from postconditions to preconditions. A natural, forward reasoning is possible by viewing a program as a mathematical function that maps one program state to another. This functional program verification technique requires a minimal mathematical background as it uses equational reasoning based on sets and functions. Thus, it can be easily taught and used in practice. In this paper, we formalize a functional program specification and verification technique and extend it for object-oriented programs. Our approach allows one to formally specify and verify the behavior of an object-oriented program in a way that is natural and closer to the way one reasons about it informally.

I. INTRODUCTION

Considering the ubiquitousness of software and the frequency of software failures, the area of correctness verification is an important part of the education of computer scientists and software engineers. There is a real concern with the lack of rigor and accountability in computer programming and software engineering [1], and the research agenda for software engineering states the need for strengthened mathematical foundation in the work force [2]. The problem extends to the state of practice in software development outside the university setting, and universities are partly responsible. The problem is not new, as shown by the following observation made in 1990 [3]:

... [there is] a fundamental difference between software engineers and other engineers. Engineers are well trained in the mathematics necessary for good engineering. Software engineers are not trained in the disciplines necessary to assure high quality software.

One of the clear differences between typical programming courses and most engineering courses is that programming courses seldom teach or make much use of mathematics [4]. Although students are exposed to logic, the topic’s treatment is quite shallow, and they are usually unable to apply logic in a practical setting, e.g., verification or reasoning about the correctness of programs. Formal program verification is not integrated in the computer science curriculum. Students may get a glimpse of it in upper-division software engineering courses in the form of Hoare-style axiomatic proof [5]. However, axiomatic proof uses backward reasoning based on the first-order predicate logic, and most students have difficulty in learning and applying it.

In this paper we formalize a functional program specification and verification technique that supports forward reasoning. The technique requires a minimal mathematical background by viewing a program as a mathematical function from one program state to another and by using equational reasoning based on sets and functions. We believe that the functional program verification technique can be effectively integrated into the standard computer science curriculum, from introductory programming courses to advanced programming and software engineering courses. It is also our conjecture that if students become proficient in functional verification, they may be able to learn the axiomatic approach easily as a complementary reasoning technique.

The basis of our work is the Cleanroom Software Engineering, a lightweight or semi-formal approach to software development, originally developed by Harlan Mills and his colleagues at IBM [6] [7]. Its name was taken from the electronics industry, where a physical clean room exists to prevent introduction of defects during hardware fabrication, and the method reflects the same emphasis on defect prevention rather than defect removal. Special methods are used at each stage of the software development—from requirement specification and design to implementation—to avoid errors. In particular, it uses specification and verification, where verification means proving, mathematically, that a program agrees with its specification [7, Chapter 4].

Our work makes two important contributions in the area of formal program specification and verification. The first contribution is a formalization of functional program specification and verification. Our work provides a solid mathematical foundation and framework for functional program verification that includes a specification notation and formal proof rules. As mentioned earlier, our technique supports natural, forward reasoning based on sets and functions. The second contribution is support for object-orientation. Our technique supports a formal verification of object-oriented programs in the presence of subclassing and dynamic dispatch. Our technique is modular in that it does not require re-specification or re-verification of an existing program when a new subclass is introduced.

The remainder of this paper is organized as follows. In Section II-A we define a subset of Java, called Jₜ, that includes...
Fig. 1. Syntax of $J_\lambda$. In the grammar, the notations $[]$ and $\{}$ denote an optional part and a 0 or more repetition, respectively.

Fig. 2. A sample $J_\lambda$ class

class IntSet {
  int[] elements;
  IntSet() {
    elements = new int[0];
  }
  int size() {
    return elements.length;
  }
  boolean has(int e) {
    boolean r = false;
    int i = 0;
    while (i < elements.length) {
      if (elements[i] == e) {
        r = true;
        i = i + 1;
      }
    }
    return r;
  }
  void insert(int e) {
    if (!has(e)) {
      int[] newElements = new int[elements.length + 1];
      int i = 0;
      while (i < elements.length) {
        newElements[i] = elements[i];
        if (newElements[i] == e) {
          newElements[i] = e;
          elements = newElements;
        }
      }
    }
  }
}

II. $J_\lambda$: A Subset of Java

In this section we define a subset of Java, called $J_\lambda$, for use as a platform for studying functional program specification and verification. We define the syntax of $J_\lambda$ formally and its semantics informally and also introduce sample $J_\lambda$ code to be used throughout this paper.

A. Syntax of $J_\lambda$

$J_\lambda$ captures essential features of imperative object-oriented programming languages. In Section III we introduce the notation for specifying the behavior of $J_\lambda$ programs formally. In particular, we describe the concurrent assignment notation for documenting both the function computed by $J_\lambda$ code and a programmer’s intention for the code. In Section IV we describe our verification technique by first defining proof rules for imperative and procedural features of $J_\lambda$ such as control statements and methods and then extending them for object-oriented features such as inheritance and method overriding. In Section V we discuss abstraction and modularity of reasoning. In Section VI we mention some of the most related work, and we conclude our paper in Section VII.

II. $J_\lambda$: A Subset of Java

In this section we define a subset of Java, called $J_\lambda$, for use as a platform for studying functional program specification and verification. We define the syntax of $J_\lambda$ formally and its semantics informally and also introduce sample $J_\lambda$ code to be used throughout this paper.

A. Syntax of $J_\lambda$

$J_\lambda$ captures essential features of imperative object-oriented programming languages such as Java. As an imperative language, it represents program states with program variables and manipulate them with control statements such as assignment statements. As an object-oriented language, it supports the following concepts as objects, classes, and inheritance. A $J_\lambda$ program consists of a set of classes, and Figure 1 shows the syntax for declaring a $J_\lambda$ class.

As in Java, a $J_\lambda$ class is a sequence of member declarations, where a member declaration can be a field declaration or a method declaration. For simplicity, only a few representative control statements are supported, such as assignment statements, if statement, while statement, and return statement. Though not shown, built-in types such as boolean and integer as well as array types are supported along with their typical operators.

A class can be defined to be a subclass of another class. A subclass inherits all the fields and methods of its superclass. It can also override inherited methods by redefining them. For a method invocation, dynamic dispatch is used as in Java based on the the runtime type of the receiver object.

B. Example

Figure 2 shows sample code written in $J_\lambda$. It defines a class IntSet, an abstraction of a set of integers. A set is represented as an array that is initialized by a constructor and manipulated by several methods such as size, has, and insert. In Section III below, we will show how to specify the behavior of the class formally.

III. Specifying $J_\lambda$

How do we specify the behavior of a program, or a section of code, written in $J_\lambda$? An execution of a $J_\lambda$ statement produces a side-effect on a program state by changing the values of some state variables such as fields and local variables. Thus, we can model a program execution as a mathematical function from one program state to another, where a program state is a mapping from state variables to their values. For example, consider the following code that swaps the values of $x$ and $y$.

\[
\begin{align*}
x &= x + y; \\
y &= x - y; \\
x &= x - y;
\end{align*}
\]
The execution of the above code can be modeled as a mathematical function that, given a program state, produces a new state in which x and y are mapped to the initial values of y and x, respectively. The rest of the state variables, if any, are mapped to their initial values; that is, their values remain the same.

How do we represent such a function? We use a concurrent assignment, a succinct notation to express a function by only stating changes in an input state. A concurrent assignment is written as $[x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n]$ and states that each $x_i$’s new value is $e_i$, evaluated concurrently in the initial state, i.e., the input state or the state just before executing the code. The value of a state variable that doesn’t appear in the left-hand side of a concurrent assignment remains the same. For example, the function that swaps two variables, $x$ and $y$, is written as $[x, y := y, x]$. We can use the concurrent assignment notation to document both the actual function computed by a section of code and our intention for the code called an intended function.

The function computed by a program is often partial in that the code works only for some well-defined input values. We extend the concurrent assignment notation to also specify such a partial function. For example, the function that swaps two variables, $x$ and $y$, is written as $[x, y := y, x]$. We can use the concurrent assignment notation to document both the actual function computed by a section of code and our intention for the code called an intended function.

In notation, it’s similar to Dijkstra’s guarded command. However, the meaning is slightly different as the conditions are evaluated sequentially from the first to the last; that is, if more than one condition hold, the function defined is the one corresponding to the first condition that holds.

A. Annotating Jₐ code

In Jₐ, we annotate the behavior of a program using the concurrent assignment notation. We write an intended function for each section of code. For example, the following is an annotated version of part of the body of the has method of the IntSet class.

```java
class IntSet{
    int[] elems;
    @elems := new int[0]
    IntSet() { /* ... */ }
    @result := elems.length
    int size() { /* ... */ }
    @result := isIn(e, elems) where
    @isin(e, a) = isIn(e, a, 0)
    @ where isIn(e, a, i)
    @ | i >= a.length -> false
    @ | else -> e == a[i] || isIn(e, a, i + 1)
    boolean has(int e) { /* ... */ }
    @has(e) -> i
    @ | else -> elems := (any a; isPerm(a, append(elems, e)))
    void insert(int e) { /* ... */ }
    @boolean isPerm(int[] a, int b[]) {
    @ int[] sa = Arrays.copyOf(a, a.length);
    @ int[] sb = Arrays.copyOf(b, b.length);
    @ Arrays.sort(sb);
    @ return Arrays.equals(sa, sb);
    }
    @int[] append(int[] a, int e) {
    @ int[] r = Arrays.copyOf(a, a.length + 1);
    @ r[a.length] = e;
    @ return r;
    }
}
```

Fig. 3. A sample Jₐ class with annotations

An annotation is preceded by an @ symbol to differentiate it from regular Jₐ code, and indentation is used to indicate the region of code that an intended function annotates.

The top-level intended function states that new value of $r$ is $isin(e,elems,0)$. A user-defined function $isin$ tests if an element (e) is contained in an array (elems) starting from a given index (0); we will show below how such a user-defined function can be defined. For a control structure such as a while statement, we specify the behavior of the whole structure as well as each of its components. We use the keyword anything to indicate that we don’t care the final value of a certain variable, typically a local or incidental variable. The symbol I denotes an identity function.

In addition to documenting the behavior of each section of code, we also describe the behavior of a whole method as its method specification. Figure 3 shows the IntSet class of which methods are annotated with intended functions. The first annotation states that the constructor initializes the field elems to an empty array. The intended function for the size method states that the method returns the length of the array elems; the pseudo variable result denotes the return value of a method. The annotation for the has method is interesting, as it is defined in terms of a user-defined function, $isin$. As shown, a user-defined function may be introduced using a where clause. The overloaded function $isin$ is defined recursively using a syntax similar to Dijkstra’s guarded command. The expression $isin(e,a,i)$ is false if $i$ is greater than or equal to $a.length$; otherwise, it is $e == a[i]$ or $isin(e,a,i+1)$. The annotation for the
To prove the correctness of \( J \), following section we will show how we can apply these rules to introduce a user-defined vocabulary for writing intended functions.

### IV. Verifying \( J_\lambda \)

#### A. Proof Rules

In the functional program verification, verifying a program is essentially comparing two mathematical functions. The core idea is to calculate the function computed by code and compare it with the intended function of the code. Figure 4 shows proof rules for representative \( J_\lambda \) language constructs.

The first rule says when a function is a correct with respect to \( (\subseteq) \) another function. This rule is needed because code often does more than what its intended function states. A function \( f_1 \) is correct with respect to \( f_2 \) if the domain of \( f_1 \) is a superset of the domain of \( f_2 \) and, for every \( x \) in the domain of \( f_2 \), both \( f_1 \) and \( f_2 \) map \( x \) to the same value. We also say \( f_1 \) is a refinement of \( f_2 \).

The next two rules are for an assignment statement and a sequential composition statement. For the assignment statement, we assume the expression on the right hand side is side-effect free. In rule R3, \( f_1; f_2 \) denotes a functional composition of two functions, \( f_1 \) and \( f_2 \), often written as \( f_2 \circ f_1 \); that is, \( (f_1; f_2)(x) = f_2(f_1(x)) \).

Rule R4 is a proof rule for an if statement and is based on a case analysis. It requires one to prove the correctness on both cases when the if condition hold and when it doesn’t.

The last rule, a proof rule for a while statement, uses an induction. To prove the correctness of a while statement with respect to an intended function \( f \), one has to prove that:

1) when the loop condition (\( E \)) doesn’t hold, an identity function (\( I \)) is correct with respect to \( f \), and
2) when the loop condition holds, the loop body (\( S \)) followed by \( f \) is correct with respect to \( f \).

Though not shown in the rule, one also need to prove the termination of the loop (see below for an example). In the following section we will show how we can apply these rules to prove the correctness of \( J_\lambda \) code.

#### B. Verifying Procedural Code

In this section we demonstrate verification of procedural code of \( J_\lambda \) that doesn’t involve objects or message sending. The following code computes the sum of all even numbers between 1 and \( n \), inclusive. As shown, it is annotated with intended functions written in terms of user-defined functions such as \( \text{isEven} \) and \( \text{sumEven} \), and intended functions are labelled with names such as \( f_0 \) and \( f_1 \).

```plaintext
@ isEven(x) = x % 2 == 0;
@ sumEven(x,y) = x > y ? 0:
@ isEven(x) = x % 2 == 0;

f0: [r := sumEven(1, n)]

i = 1;
int i = 1;
r = 0;

f1: [r, i := r + sumEven(i, n), anything]

while (i <= n) {
    if (i % 2 == 0)
        r = r + i;
    i = i + 1;
}
```

The verification of the above code involves discharging the following four proof obligations.

1) \( f_1; f_2 \subseteq f_0 \), i.e., proof that \( f_1 \) followed by \( f_2 \) is a correct implementation of \( f_0 \).
2) Correctness of \( f_1 \) and its code
3) Correctness of \( f_2 \) and its code, which involves the following three sub-proofs.
   a) Termination of the loop
   b) Basis step: \( \neg(i \leq n) \Rightarrow I \subseteq f_2 \)
   c) Induction step: \( i \leq n \Rightarrow f_3; f_2 \subseteq f_2 \)
4) Correctness of \( f_3 \) and its code

Below we show the proofs of these obligations. We first prove \( f_1; f_2 \subseteq f_0 \).

\[
f_1; f_2 = [r, i := 0, 1]; [r, i := r + \Sigma_e(i, n), \bot]
\]

\[
= [r, i := 0 + \Sigma_e(1, n), \bot]
\]

\[
= [r, i := \Sigma_e(i, n), \bot]
\]

\[
\subseteq [r := \Sigma_e(i, n)]
\]

\[
= f_0
\]

where \( \Sigma_e(x, y) \) denotes the sum of all even numbers from \( x \) to \( y \), inclusive, and \( \bot \) denotes an arbitrary value.

The proof of the second obligation, \( f_1 \) and its code, is trivial.
We next prove \( f_2 \), i.e., the correctness of the \( \textbf{while} \) statement in terms of its intended function \( (f_2) \) and that of the loop body \( (f_3) \); the proof of the loop body is done separately (see below). First, we prove the termination of the loop by finding a loop variant, \( n - i \). We note that, as stated in \( f_3 \), the value of \( i \) increases by 1 upon each iteration of the loop. Thus, the loop variant decreases by 1 on each iteration, eventually terminating the loop when it becomes negative. We next prove the correctness of the loop inductively. We first prove the basis step: \( \neg (i \leq n) \Rightarrow I \subseteq f_2 \). When \( i \) is greater than \( n \), we have the following.

\[
f_2 = [r, i := r + \Sigma_e(i, n), \bot] = [r, i := r + 0, \bot] = [r, i := r, \bot] \supseteq [r, i := r, i] = I
\]

We next prove the induction step: \( i \leq n \Rightarrow f_3 \subseteq f_2 \). When \( i \) is less than or equal to \( n \), we have the following.

\[
f_3; f_2 = \begin{cases} \text{isEven}(i) \rightarrow r, i := r + i, i + 1 \\ \text{else} \rightarrow i := i + 1; [r, i := r + \Sigma_e(i, n), \bot] \end{cases} = [r, i := r + \text{isEven}(i) \land i: 0 + \Sigma_e(i + 1, n), \bot] = [r, i := r + \Sigma_e(i, n), \bot] = f_2
\]

This concludes the proof of \( f_2 \) and its code.

The last obligation is to prove the refinement of \( f_3 \), i.e., the correctness of the loop body. The proof is straightforward, as shown below.

\[
f_3 = \begin{cases} \text{isEven}(i) \rightarrow r, i := r + i, i + 1 \\ \text{else} \rightarrow i := i + 1; [r, i := r + \Sigma_e(i, n), \bot] \end{cases} = [r, i := r + \text{isEven}(i) \land i: 0 + \Sigma_e(i + 1, n), \bot] \supseteq [r, i := r + \Sigma_e(i, n), \bot] = f_2
\]

C. Verifying Methods and Method Invocations

For a modular verification, it is essential to verify the correctness of a method just once. To achieve this, we use the specification of the method for the verification of a method invocation. For this, we first have to verify that a method body is correct with respect to the intended function of the method. However, there is nothing new about this verification except that, for the verification purpose, we can view a return statement as an assignment to a pseudo variable, result, as follows; recall that we use result to denote the return value of a method in the method specification.

\[
\begin{align*}
8 &| \text{result} := x + y \\
\text{return} & x + y
\end{align*}
\]

The next step is to verify a method invocation. As mentioned earlier, we use the specification—intended function—of the invoked method with a proper renaming (see Section IV-D for determining the invoked method in the presence of dynamic method dispatch). This approach is modular in that a method implementation needs to be verified just once and for all. Any change in the implementation doesn’t invalidate the verification of client code as long as the method specification remains the same. The following two axioms are for verifying a method invocation in two different contexts.

\[
\text{R6A: } E_0.m(E_1, \ldots, E_n) \subseteq f_m(E_1/x_1, \ldots, E_n/x_n, E_0/x/result)
\]

\[
\text{R6B: } x = E_0.m(E_1, \ldots, E_n) \subseteq f_m(E_1/x_1, \ldots, E_n/x_n, E_0/this, x/result)
\]

where \( f_m \) is the intended function of \( m, x_1, \ldots, x_n \) are formal parameters of \( m \), and \( x/y \) denotes the renaming of every free occurrence of \( y \) with \( x \). Note that pseudo variables such as this and result in the intended function are appropriately renamed. Refer to Section IV-D3 for an example verification of a method invocation in object-oriented code.

D. Verifying Object-Oriented Code

1) Problem: How to verify object-oriented code in the presence of subclassing? The verification of a single class in isolation doesn’t introduce a new problem. However, a new subclass may change the behavior of existing code because of method overriding and dynamic dispatch, and thus may require re-verification of existing code.

As an example, consider introducing a new subclass of the IntSet class, say SortedIntSet, that stores the elements sorted. The new subclass overrides the insert method to add a new element and keep all elements sorted. It may also introduce additional methods, such as first that returns the first element of the set, to observe the order of the elements. Now, let’s assume that the correctness of IntSet has already been verified. When a new class, e.g., SortedIntSet, is introduced as a subclass of IntSet, do we need to verify only the additional method introduced in a SortedIntSet or do we also need to reverify all the methods inherited from IntSet such as the has method. It is well known that we have to reverify or re-test all the methods of IntSet because of method overriding and dynamic dispatch. For example, if the IntSet class has a method named insertAll that adds multiple elements and is written in terms of the insert method, we have to reverify the inherited insertAll method because its behavior may have been changed; it uses the insert method which is overridden in SortedIntSet and because of dynamic dispatch the overriding method will be invoked in the context of SortedIntSet. The same is true for client code that uses IntSet. In short, we have to reverify any code that uses the overridden method directly or indirectly.

2) Approach: We use a behavioral notion of subtyping to support a modular verification of object-oriented code in the presence of subclassing and dynamic dispatch. An object of a subclass must behave like an object of its superclass [8]. As before, we verify client code using the specification of an invoked method, where the invoked method is determined.
statically based on the declared, or static, types. However, one difference is that whenever a new subclass is introduced, it has to be proved to be a behavioral subtype of its superclass. In particular, every overriding method has to be proved to behave like its overridden method. If this is done, the existing verification of the class or its client code is still valid because the verification was done by using the specification of the overridden method and the overriding method preserves the specification by behaving like the overridden method; the overriding method is correct with respect to the specification of the overridden method. The approach is modular in that we only need to verify new code introduced in the subclass; we don’t have to reverify any existing code.

Below we formulate the notion of behavioral subtyping in terms of intended functions. Let $S$ be a subclass of $T$. We say $S$ is a behavioral subtype of $T$ if for every method $m$ that is overridden in $S$, the following two conditions are satisfied:

- $\text{dom}(f^S_m) \supseteq \text{dom}(f^T_m)$, where $f^S_m$ and $f^T_m$ are intended functions of $m$ in $S$ and $T$, respectively
- $f^S_m(x) = f^T_m(x)$ for each $x \in \text{dom}(f^T_m)$

The first condition states that an overriding method accepts all the values accepted by its overridden method, and the second condition states that the overriding method produces the same value as that of the overridden method.

In the following subsection, we show a sample verification of a behavioral subtyping relationship.

3) Example: As explained earlier, a verification of object-oriented code generally consists of two steps. The first step is to verify the code itself. If the code is a client of some class, the code is verified using the specification of the class that is determined statically based on type declarations. The second step is to verify, for each subclass, a behavioral subtyping relationship between it and its superclass.

As an example, consider the following code that uses the IntSet class introduced in earlier sections.

```java
@ | s.has(1) -> result := s.size()
| else -> elms := s.size() + 1,
| @ | any a; s.inserted(a, s.append(s.elems, 1)))
int addOneAndReturnSize(IntSet s) {
  @f1: [s.has(1) -> I
  @ | else -> elms := (any a; s.isPerm(a, s.append(s.elems, 1))))
  s.insert(1);
  @f2: [result := s.size()]
  return s.size();
}
```

The verification of this client code is straightforward. We need to prove $f_1 : f_2 \subseteq f_0$ and the correctness of $f_1$ and $f_2$. Note that for the verification of $f_1$ we use the specification of the IntSet class, the declared type of the formal parameter $s$. We use the Rule R6A (see Section IV-C) and replace the method call, $s.insert(1)$, with its specification after an appropriate renaming for formal parameters and the implicit this, i.e., $f$ for $e$ and $s$ for this.

Let us now introduce the SortedIntSet class as a subclass of the IntSet class. Figure 5 shows the definition of SortedIntSet along with its specification. In addition to proving the correctness of SortedIntSet, we also need to prove a behavioral subtyping relationship between SortedIntSet and IntSet. If this is done, the verification of the client code of IntSet, such as the addOneAndReturnSize method, will be still valid when an object of SortedIntSet is substituted for an object of IntSet, e.g., for the formal parameter $s$. For the verification of subtyping, we need to consider only the insert method that is overridden. The specifications of this method in both classes are shown below.

```java
@ | has(e) -> I
| else -> elms := (any a; isPerm(a, append(elms, e)))
void insert(int e) {
  int[] newElems = new int[elems.length + 1];
  int i = 0;
  int j = 0;
  boolean inserted = false;
  while (j <= elems.length) {
    if (!inserted && e < elems[j]) {
      newElems[i] = e;
      inserted = true;
    } else {
      newElems[i] = elems[j];
      j = j + 1;
    }
  i = i + 1;
  if (!inserted) {
    newElems[i] = e;
  }
  elems = newElems;
}
}
```

Fig. 5. Class SortedIntSet

Both functions are total, and $f_2$ maps the elms field to an array that $f_1$ maps to; a sorted array is a permutation of the original array. Thus, SortedIntSet is a behavioral subtype of IntSet.

V. ABSTRACTION AND MODULAR REASONING

In Section IV-D3, when we reasoned about the correctness of client code of a class, we used the representation, or concrete, value of the class. For example, the intended functions

\[ f_1 : f_2 \subseteq f_0 \]

\[ f_2 : \text{SortedIntSet} \]

\[ f_2 : \text{IntSet} \]

\[ f_2 : \text{addOneAndReturnSize} \]

\[ f_2 : \text{insert} \]

\[ f_2 : \text{isPerm} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{Arrays.equals} \]

\[ f_2 : \text{Collections} \]

\[ f_2 : \text{Collections.sort} \]

\[ f_2 : \text{Collections.stream} \]

\[ f_2 : \text{IntSet} \]

\[ f_2 : \text{SortedIntSet} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{isPerm} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{Collections} \]

\[ f_2 : \text{Collections.sort} \]

\[ f_2 : \text{Collections.stream} \]

\[ f_2 : \text{IntSet} \]

\[ f_2 : \text{SortedIntSet} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{isPerm} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{Collections} \]

\[ f_2 : \text{Collections.sort} \]

\[ f_2 : \text{Collections.stream} \]

\[ f_2 : \text{IntSet} \]

\[ f_2 : \text{SortedIntSet} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{isPerm} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{Collections} \]

\[ f_2 : \text{Collections.sort} \]

\[ f_2 : \text{Collections.stream} \]

\[ f_2 : \text{IntSet} \]

\[ f_2 : \text{SortedIntSet} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{isPerm} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{Collections} \]

\[ f_2 : \text{Collections.sort} \]

\[ f_2 : \text{Collections.stream} \]

\[ f_2 : \text{IntSet} \]

\[ f_2 : \text{SortedIntSet} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{isPerm} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{Collections} \]

\[ f_2 : \text{Collections.sort} \]

\[ f_2 : \text{Collections.stream} \]

\[ f_2 : \text{IntSet} \]

\[ f_2 : \text{SortedIntSet} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{isPerm} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{Collections} \]

\[ f_2 : \text{Collections.sort} \]

\[ f_2 : \text{Collections.stream} \]

\[ f_2 : \text{IntSet} \]

\[ f_2 : \text{SortedIntSet} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{isPerm} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{Collections} \]

\[ f_2 : \text{Collections.sort} \]

\[ f_2 : \text{Collections.stream} \]

\[ f_2 : \text{IntSet} \]

\[ f_2 : \text{SortedIntSet} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{isPerm} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{Collections} \]

\[ f_2 : \text{Collections.sort} \]

\[ f_2 : \text{Collections.stream} \]

\[ f_2 : \text{IntSet} \]

\[ f_2 : \text{SortedIntSet} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{isPerm} \]

\[ f_2 : \text{append} \]

\[ f_2 : \text{Collections} \]

\[ f_2 : \text{Collections.sort} \]

\[ f_2 : \text{Collections.stream} \]
of the \texttt{addOneAndReturnSize} method and the \texttt{insert} method call statement were written in terms of the \texttt{elems} field of the \texttt{IntSet} class. Such a implementation-dependent specification is often hard to read, understand and reason about because one has to know particular implementation details—e.g., the use of an array to store the elements of a set—and manipulate low-level concrete, representation values—e.g., an array. Worse, such a specification and reasoning is not modular in that if the representation of the class changes the specification and reasoning of its client code should be re-done in terms of the new representation.

One possible solution would be to have multiple specifications of different abstraction levels, e.g., a public specification for clients and a private specification for the implementation. The public specification is written in terms of abstract values, whereas a private specification is written in terms of concrete representation values. For example, the following listing shows a public specification of the \texttt{IntSet} class, written in terms of a mathematical set.

```java
class IntSet {
    @ with map(elems, 0) where
    @ map(a, i) = i => a.length ? ∅ : a[i] ∪ map(a, i+1)

    int[] elems;

    @this := ∅
    IntSet() { /* ... */ }

    @result := |this|
    Int size() { /* ... */ }

    @result := e ∈ this
    boolean has(int e) { /* ... */ }

    @e ∈ this -> I | else -> this := this ∪ {e}
    void insert(int e) { /* ... */ }
}
```

An \texttt{IntSet} object is now viewed abstractly as a mathematical set, and the intended functions for the \texttt{IntSet} class are written using the vocabulary of sets such as ∅, ∈, and ∪. The first annotation, the \texttt{with} clause, defines an abstraction function that maps a concrete representation value (i.e., an array) to an abstract specification value (i.e., a set). It allows to reason about the correctness of a (public) specification written in terms of abstract values. Note that the annotation for a method body can still be written in terms of concrete representation values. For example, the following listing shows a public specification of the \texttt{IntSet} class, written in terms of a mathematical set.

```java
class SortedIntSet extends IntSet
    @ with map(this) where
    @ map(s) = |s| == 1 ? ∅ : first(s) ∪ rest(s) { /* ... */ }
}
```

VI. RELATED WORK

The importance of mathematics to the computer program development has long been recognized since Hoare’s seminal work on an axiomatic approach to defining the meanings of programs and proving their correctness [5], and there have been numerous publications on this subject. Below we mention some of the most related work.

For formal specifications of programs, we took the ideas from Cleanroom [6] [9] [7]—e.g., functional semantics and intended functions—and enhanced them with recent advances in formal behavioral interface specification languages (BISL), especially design-by-contract (DBC) notations [10] and such BISLs as JML [11]. As in Cleanroom, intended functions are expressed in concurrent assignment statements. However, following the idea of DBC, Java’s expression syntax is used to write intended functions. This makes it easy for programmers to learn and write intended functions, as it eliminates or minimizes the overhead of learning a separate specification notation. Some formal specification languages provide an extension mechanism to introduce a user-defined vocabulary for writing specifications. In JML, for example, one can introduce specification-only methods, called model methods, for writing specifications [12]. In JML, one can introduce model methods as well as user-defined mathematical functions to enrich the vocabulary for writing intended functions.

JML inspired the design of Jδ on supporting abstract specifications and modular reasoning. There are dual uses of a method specification—for verifications of client code and the method implementation. In JML, these are supported by model variables and privacy of specifications. A model variable is a specification-only variable to describe the abstract state of some program variables [12]. An abstract method specification can be written by referring to and manipulating the abstract
values of model variables. This abstract specification is used to verify the client code. However, when reasoning about the correctness of a method implementation, one has to know how the abstract state is represented. For this, a model variable may be accompanied by the specification of an abstraction function, that says how to map concrete program states to the abstract values of model variables [13] [14]. For modular reasoning of client code, JML also supports privacy of specifications [11]. A public specification, for example, can't be written by referring to private information such as implementation decisions and details that are irrelevant to reasoning about client code.

We found no published work on extending Cleanroom-style functional specification and verification for object-oriented programs, except for Ferrer’s [15]. Ferrer proposed to specify the behavior of a class in object-oriented programs by writing the intended functions of mutation methods in terms of the observer methods of the class [15]. However, such a specification has an algebraic flavor and will be inherently incomplete, as the intended function of the observer methods themselves can’t be written. In J3, a complete specification can be written by referring to and manipulating abstract values, and it has a flavor of model-oriented specifications.

The notion of behavioral subtyping is a key to a modular reasoning and verification of object-oriented programs [8]. We adapted the work of Leavens [16] on verification logic for object-oriented programs that relies on a behavioral notion of subtyping. In essence, the verification of client code is the same as in procedural programs. For each subclass, however, one has to prove that it is a behavioral subtype of its superclass by showing that each overriding method behaves like the overridden method. This approach reflects the way programmers reason informally about object-oriented programs, in that it allows them to use static type information, which avoids the need to consider all possible runtime subtypes. Our specific adaptation is to define a behavioral notion of subtyping in terms of intended functions for use in the functional verification.

VII. CONCLUSION

As a verification and validation method for software systems, formal reasoning—reasoning based on mathematics—is an aid to success in preparing students to develop correct and reliable software systems. As the first step for integrating a formal program verification to the standard computer science curriculum, from introductory programming courses to advanced programming and software engineering courses, we formalized a functional program verification technique by defining a specification notation and proof rules. We also extended the technique to support a modular verification of object-oriented programs. In essence, our technique reflects the way programmers reason informally about object-oriented programs by requiring to verify that each subclass behaves like its superclass.

Our work provides a solid mathematical foundation for the functional program specification and verification. We strongly believe that our technique can be effectively integrated into the standard computer science curriculum because, unlike Hoare logic, it supports forward reasoning and requires a minimal mathematical background by viewing a program as a mathematical function from one program state to another and by using equational reasoning based on sets and functions.

ACKNOWLEDGMENT

The work of the author was supported in part by NSF grants CNS-0707874 and DUE-0837567.

REFERENCES