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.

A short version of this report will appear in the *Annual International Conference on Software Engineering (SE 2010)*, December 6-7, 2010, Phuket, Thailand.

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 the 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 the 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 J_λ, that includes

```
⟨classDecl⟩ → class ⟨ident⟩ [extends ⟨ident⟩]
  '{' {⟨memberDecl⟩} '}'
⟨memberDecl⟩ → ⟨fieldDecl⟩ | ⟨methDecl⟩
⟨fieldDecl⟩ → ⟨varDecl⟩ [= ⟨expr⟩] ;
⟨methDecl⟩ → [⟨type⟩] ⟨ident⟩ ([⟨paramDecls⟩])
  '{' {⟨stmts⟩} '}'
⟨paramDecls⟩ → ⟨varDecl⟩ {, ⟨varDecl⟩}
⟨varDecl⟩ → ⟨type⟩ ⟨ident⟩
⟨stmts⟩ → {⟨stmt⟩}
⟨stmt⟩ → ⟨varDecl⟩ [= ⟨expr⟩] ;
  | ⟨assignStmt⟩
  | ⟨ifStmt⟩
  | ⟨whileStmt⟩
  | ⟨returnStmt⟩
  | '{' ⟨stmts⟩ '}'
⟨assignStmt⟩ → ⟨expr⟩ = ⟨expr⟩ ;
⟨ifStmt⟩ → if ( ⟨expr⟩ ) ⟨stmt⟩ [else ⟨stmt⟩]
⟨whileStmt⟩ → while ( ⟨expr⟩ ) ⟨stmt⟩
⟨returnStmt⟩ → return [⟨expr⟩] ;
⟨expr⟩ → ⟨ident⟩
  | ⟨expr⟩.⟨expr⟩
  | ⟨expr⟩.⟨ident⟩([⟨expr⟩ {, ⟨expr⟩}])
  | ...
```

Fig. 1. Syntax of J_λ. In the grammar, the notations [] and {} denote an optional part and a 0 or more repetition, respectively.

essential features of object-oriented programming languages. In Section III we introduce the notation for specifying the behavior of J_λ programs formally. In particular, we describe the concurrent assignment notation for documenting both the function computed by J_λ 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_λ 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_λ: A Subset of Java

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

A. Syntax of J_λ

J_λ 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 support such concepts as objects, classes and inheritance. A J_λ program consists of a set of classes, and Figure 1 shows the syntax for declaring a J_λ class.

As in Java, a J_λ 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 statement, **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.

```
class IntSet {
  int[] elems;

  IntSet() {
    elems = new int[0];
  }

  int size() {
    return elems.length;
  }

  boolean has(int e) {
    boolean r = false;
    int i = 0;
    while (i < elems.length) {
      if (elems[i] == e) {
        r = true;
      }
      i = i + 1;
    }
    return r;
  }

  void insert(int e) {
    if (!has(e)) {
      int[] newElems = new int[elems.length + 1];
      int i = 0;
      while (i < elems.length) {
        newElems[i] = elems[i];
      }
      newElems[i] = e;
      elems = newElems;
    }
  }
}
```

Fig. 2. A sample J_λ class

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_λ. 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_λ

How do we specify the behavior of a program, or a section of code, written in J_λ? An execution of a J_λ 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`.

```
x = x + y;
y = x - y;
x = x - y;
```

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, $[n! = 0 \rightarrow avg := sum/n]$ is a state changing function that is defined only for a state in which x is not zero. In a conditional concurrent assignment such as above, the condition is evaluated in the initial state. We often want to specify different functions based on some conditions. For example, the following intended function determines the sign of the variable `x`.

```
[x > 0 -> sign := 1
 | x < 0 -> sign := -1
 | else -> sign := 0]
```

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 J_λ code

In J_λ, 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.

```
@[r := isIn(e, elems, 0)]
  @[i := 0]
    int i = 0;

  @[r, i := r || isIn(e, elems, i), anything]
    while (i < elems.length) {
      @[r, i := r || elems[i] == e, i + 1]
        @[elems[i] == e -> r := true | else -> I]
          if (elems[i] == e) {
              @[r := true]
                r = true;
          }

        @[i := i + 1]
          i = i + 1;
    }
```

```
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(sa);
  @   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 J_λ class with annotations

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

The top-level intended function states that the 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

`insert` method is also interesting. It states that if the element is already contained in the set (i.e., `has(e)`), then the method has no effect as indicated by an identity function, **I**; otherwise, the new value of `elems` is any permutation of the old value of `elems` appended with the element `e`. The expression (**any** x; $B(x)$) introduces a loose specification by denoting an arbitrary value x that satisfies the Boolean expression $B(x)$. The notion of permutation is defined by introducing a specification-only method `isPerm` that tests whether an array is a permutation of another; the definition appears inside an annotation, indicating that it can be used only in annotations but not in program code. This kind of specification-only methods is another way to introduce a user-defined vocabulary for writing intended functions.

IV. Verifying J_λ

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_λ language constructs.

The first rule says when a function is a *correct with respect to* ($\sqsubseteq$) 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 that 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_λ code.

B. Verifying Procedural Code

In this section we demonstrate a verification of procedural code of J_λ 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 `isEven` and `sumEven`, and intended functions are labelled with names such as f_0 and f_1.

$$\text{R1: } \frac{\text{dom}(f_1) \supseteq \text{dom}(f_2),\ f_1(x) = f_2(x) \text{ for each } x \in \text{dom}(f_2)}{f_1 \sqsubseteq f_2}$$

$$\text{R2: } x = E \sqsubseteq [x := E]$$

$$\text{R3: } \frac{S_1 \sqsubseteq f_1,\ S_2 \sqsubseteq f_2,\ f_1; f_2 \sqsubseteq f}{S_1; S_2 \sqsubseteq f}$$

$$\text{R4: } \frac{E \Rightarrow S_1 \sqsubseteq f,\ \neg E \Rightarrow S_2 \sqsubseteq f}{\textbf{if}\,(E)\,S_1\,\textbf{else}\,S_2 \sqsubseteq f}$$

$$\text{R5: } \frac{\neg E \Rightarrow I \sqsubseteq f,\ E \Rightarrow S; f \sqsubseteq f}{\textbf{while}\,(E)\,S \sqsubseteq f}$$

Fig. 4. Proof rules for representative J_λ constructs

```
@ isEven(x) = x % 2 == 0;
@ sumEven(x,y) = x > y ? 0:
@   sumEven(x+1,y) + (isEven(x) ? x : 0);

@f0: [r := sumEven(1, n)]
  @f1: [r, i = 0, 1]
   int i = 1;
   r = 0;

  @f2: [r, i := r + sumEven(i, n), anything]
   while (i <= n) {
     @f3: [isEven(i) -> r, i := r + i, i + 1
         | else -> i := i + 1]
      if (i % 2 == 0)
        r = r + i;
      i = i + 1;
  }
```

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

1) $f_1; f_2 \sqsubseteq f_0$, 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: $\neg(i \le n) \Rightarrow I \sqsubseteq f_2$
   c) Induction step: $i \le n \Rightarrow f_3; f_2 \sqsubseteq f_2$
4) Correctness of f_3 and its code

Below we show the proofs of these obligations. We first prove $f_1; f_2 \sqsubseteq f_0$.

$$\begin{aligned} f_1; f_2 &= [r, i := 0, 1]; [r, i := r + \Sigma_e(i, n), \bot] \\ &= [r, i := 0 + \Sigma_e(1, n), \bot] \\ &= [r, i := \Sigma_e(i, n), \bot] \\ &\sqsubseteq [r := \Sigma_e(i, n)] \\ &= f_0 \end{aligned}$$

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 **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: $\neg(i \leq n) \Rightarrow I \sqsubseteq f_2$. When i is greater than n, we have the following.

$$\begin{aligned} f_2 &= [r, i := r + \Sigma_e(i, n), \bot] \\ &= [r, i := r + 0, \bot] \\ &= [r, i := r, \bot] \\ &\sqsupseteq [r, i := r, i] = I \end{aligned}$$

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

$$\begin{aligned} f_3; f_2 &= [\text{isEven}(i) \rightarrow r, i := r + i, i + 1 \\ &\quad \mid \textbf{else} \rightarrow i := i + 1]; [r, i := r + \Sigma_e(i, n), \bot] \\ &= [r, i := r + (\text{isEven}(i)\,?\,i : 0) + \Sigma_e(i + 1, n), \bot] \\ &= [r, i := r + \Sigma_e(i, n), \bot] \\ &= f_2 \end{aligned}$$

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.

$$\begin{aligned} f_3 &= [\text{isEven}(i) \rightarrow r, i := r + i, i + 1 \mid \textbf{else} \rightarrow i := i + 1]; \\ &= [\text{isEven}(i) \rightarrow r := r + i \mid \textbf{else} \rightarrow I]; [i := i + 1] \\ &\sqsupseteq \textbf{if } (\text{i \% 2 == 0}) \text{ r = r + i;} \\ &\quad \text{i = i + 1;} \end{aligned}$$

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.

```
@[ result := x + y ]
return x + y;
```

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.

R6A: $E_0.m(E_1, \ldots, E_n) \sqsubseteq f_m(E_1/x_1, \ldots, E_n/x_n, E_0/this)$

R6B: $$\begin{aligned} &x = E_0.m(E_1, \ldots, E_n) \\ \sqsubseteq\ &f_m(E_1/x_1, \ldots, E_n/x_n, E_0/this, x/result) \end{aligned}$$

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 store all the 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 `SortedIntSet` is introduced as a subclass of `IntSet`, do we need to verify only the additional methods introduced in `SortedIntSet` such as the overriding `insert` method 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 method 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.

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

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.

```
@f0: [s.has(1) -> result := s.size()
@     | else -> result, s.elems := s.size() + 1,
@         (any a; s.isPerm(a, s.append(s.elems, 1)))]
int addOneAndReturnSize(IntSet s) {
  @f1: [s.has(1) -> I
  @     | else -> s.elems :=
  @         (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 $f_1; f_2 \sqsubseteq f_0$ 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., 1 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.

```
class SortedIntSet extends IntSet {
  @[elems := new int[0]]
  SortedIntSet() {
    super();
  }

  @[has(e) -> I
  @ | else -> elems := (any a; sortInserted(a, elems, e))]
  void insert(int e) {
    if (!has(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;
    }
  }

  @ boolean sortInserted(int[] a, int[] b, int e) {
  @   int[] sb = append(b, e);
  @   Arrays.sort(sb);
  @   return Arrays.equals(a, sb);
  @ }
}
```

Fig. 5. Class `SortedIntSet`

```
f1: IntSet
[has(e) -> I
 | else -> elems := (any a; isPerm(a, append(elems,e)))]

f2: SortedIntSet
[has(e) -> I
 | else -> elems := (any a; sortInserted(a, elems, e))]
```

Both functions are total[1], and f_2 maps the `elems` 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

[1]Technically, f_1 and f_2 are not functions but relations because the **any** expression introduces a loose specification. For relations f_1 and f_2, we need to show $f_2(x) \subseteq f_1(x)$ for each $x \in \mathrm{dom}(f_1)$ to prove $f_2 \sqsubseteq f_1$.

of the addOneAndReturnSize method and the insert method call statement were written in terms of the elems field of the IntSet class. Such an 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 IntSet class, written in terms of a mathematical set.

```
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 IntSet object is now viewed abstractly as a mathematical set, and the intended functions for the IntSet class are written using the vocabulary of sets such as $\emptyset$, $\in$, and $\cup$. The first annotation, the **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 representation values, e.g., by referring the elems field. A mathematical model such as a set can be provided as a built-in mathematical library or introduced as a user-defined library as needed.

The listing below shows the annotation of the addOneAndReturnSize method rewritten by referring to the public specification of the IntSet class.

```
@[1 ∈ s -> result := |s|
@    | else -> result, s := |s| + 1, s ∪ {e}]
int addOneAndReturnSize(IntSet s) {
  @[1 ∈ s -> I | else -> s := s ∪ {e}]
  s.insert(1);

  @[result := |s|]
  return s.size();
}
```

As the annotation now doesn't depend on a particular implementation detail or decision of the IntSet class, e.g., the use of an array, it doesn't have to be re-specified or re-verified when the implementation of the IntSet class changes as long as its interface and specified behavior remain the same.

When the abstract values of a subclass is different from those of its superclass, we also need to define a function, similar to an abstraction function, to map the values of a subclass to those of its superclass. This function, called a *coercion function*, coerces a value of a subclass to its superclass and allows one to reason about a behavioral subtyping relationship between a subclass and its superclass; the intended functions of an overridden method and an overriding methods may be written using different abstract values. The annotation below shows a coercion function for the SortedIntSet to map a sequence, an abstraction of the SortedIntSet, to a set.

```
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 J_λ, 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 J_λ 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 J_λ, 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

[1] D. Gries, "Improving the curriculum through the teaching of calculation and discrimination," in *Teaching and Learning Formal Methods*, C. N. Dean and M. G. Hinchey, Eds. Academic Press, 1996, pp. 181–196.

[2] CSTB, "Scaling up: a research agenda for software engineering," *Communications of the ACM*, vol. 33, no. 3, pp. 281–293, Mar. 1990, by Computer Science and Technology Board, National Research Council.

[3] J. C. Cherniavsky, "Software failures attract congressional attention," *Computing Research News*, vol. 2, no. 1, pp. 4–5, Jan. 1990.

[4] D. L. Parnas, "Teaching programming as engineering," in *Software Fundamentals*, D. M. Hoffman and D. M. Weiss, Eds. Addison-Wesley, 2001, pp. 579–592.

[5] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580,583, Oct. 1969.

[6] H. D. Mills, M. Dyer, and R. Linger, "Cleanroom software engineering," *IEEE Software*, vol. 4, no. 5, pp. 19–25, Sep. 1987.

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

[8] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811–1841, Nov. 1994.

[9] R. Oshana, "Tailoring Cleanroom for industrial use," *IEEE Software*, vol. 15, no. 6, pp. 46–55, Nov. 1998.

[10] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[11] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, Mar. 2006.

[12] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards, "Model variables: Cleanly supporting abstraction in design by contract," *Software—Practice & Experience*, vol. 35, no. 6, pp. 583–599, May 2005.

[13] C. A. R. Hoare, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, no. 4, pp. 271–281, 1972.

[14] B. Liskov and J. Guttag, *Program Development in Java*. Cambridge, Mass.: The MIT Press, 2001.

[15] G. J. Ferrer, "Teaching Cleanroom software engineering with object-oriented data abstraction," *Journal of Computer Sciences in Colleges*, vol. 21, no. 5, pp. 155–161, 2006.

[16] G. T. Leavens, "Modular specification and verification of object-oriented programs," *IEEE Software*, vol. 8, no. 4, pp. 72–80, Jul. 1991.