CS 156 πVC Tutorial

Jason Auerbach
jasonaue@stanford.edu

Joel Galenson
joel486@stanford.edu

Introduction

πVC is a verifying compiler. It compiles annotated (otherwise known as specified) programs written a C-like language called *pi* (short for "prove it"). Specified programs contain the standard program text, as well as additional text that is used to specify properties of the program at various points in its execution. piVC uses this additional text to attempt to prove that the specified properties hold for all possible inputs to the program.

Running and Using πVC

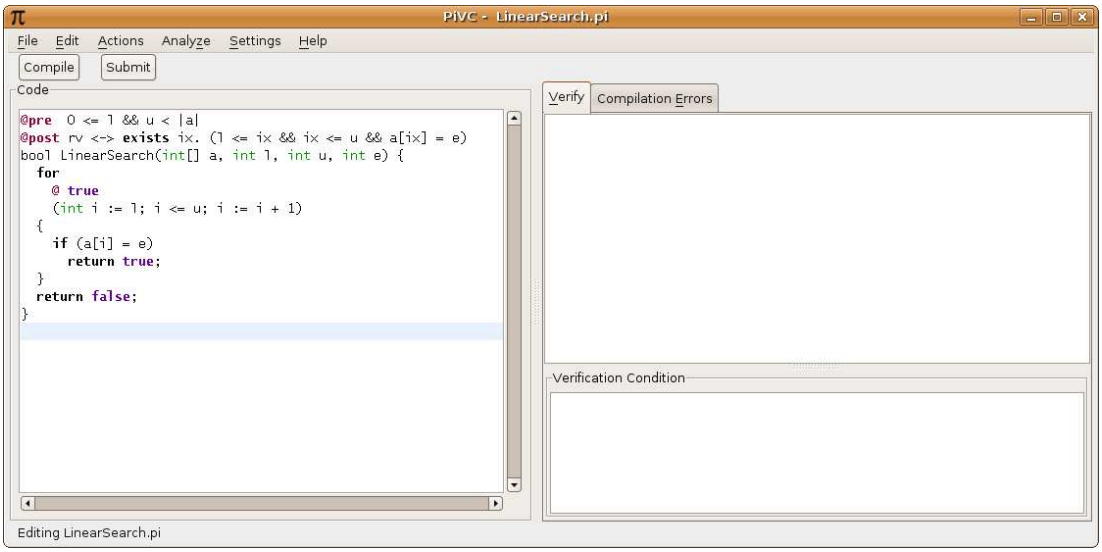
The πVC client can be downloaded from the CS156 website. It is an OS-independent JAR file. You probably already have Java installed on your machine, but if you don't, you will need to install Java before using πVC. To launch πVC, type the command `java -jar path_to_jar_file` into a terminal, or, depending on your operating system, you can probably just double-click on the JAR file. πVC uses a client/server relationship. Whenever you compile a program, the compilation request is sent to a centralized server, which then responds with the results of the compilation. Thus, to compile a πVC program, you must be connected to the internet.

πVC is easy to use (although hard to master!). Simply open and start typing a Pi program in the left pane. When you compile the program, if there are no compile errors, πVC will generate all the verification conditions (VCs) in your code and see how many are valid. This information will be given to you in the `Verify` tab in the right pane. Each basic path has a node in the tree. The node is green if the corresponding VC is valid, red if it is invalid, and yellow if it is outside of the $T_{\mathbb{Z}} \cup T_A$ fragment (you will likely never run into this). By expanding a basic path node, you can see a list of the exact steps in the path. By clicking on a basic path, the corresponding VC will be displayed in the `Verification Condition` pane.

The website for πVC is http://cs.stanford.edu/~jasonaue/pivc. This website contains several sample programs, and the solutions to selected samples.

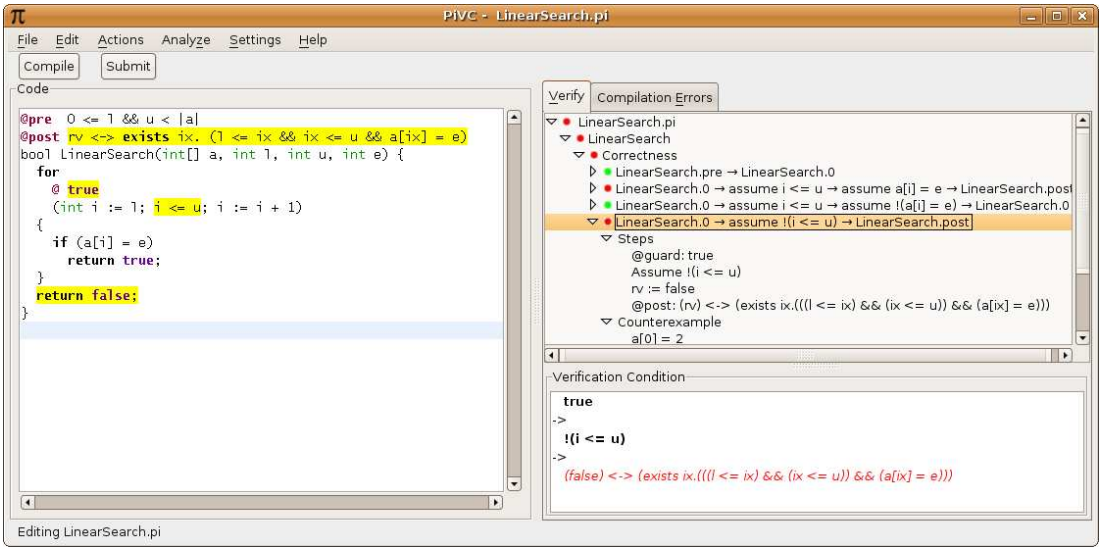
An example of using πVC

We will present a simple example of using πVC to verify the `LinearSearch` program. First start πVC as shown above and then open the `LinearSearch.pi` file (which can be downloaded from the πVC website).



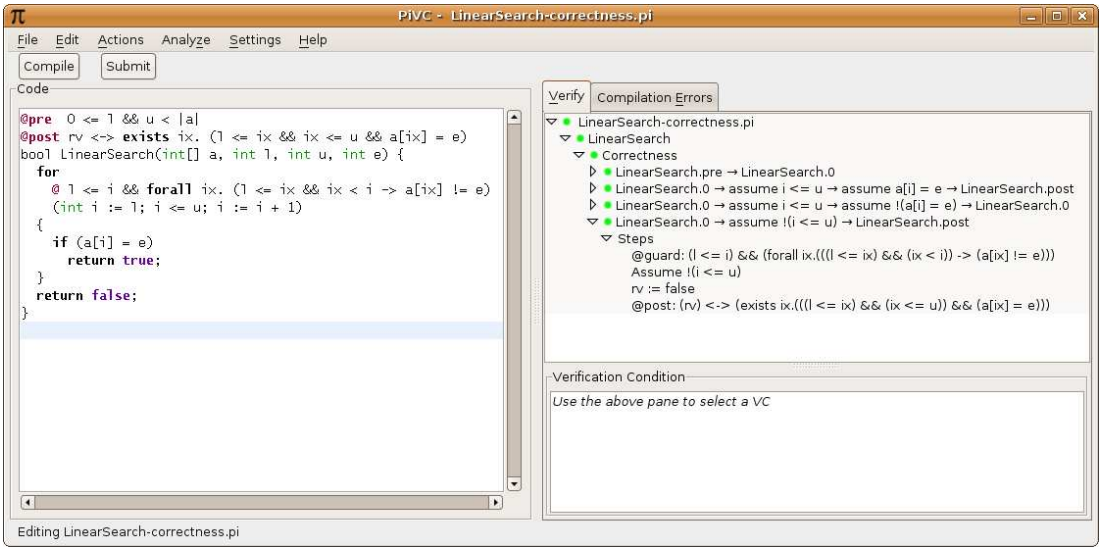
We see that the function specification is provided for us. The postcondition specifies that the function returns true if and only if the element e exists in the array a between the bounds of $[l, u]$.

Let's first try compiling it and seeing what happens. It looks like not all the verification conditions are valid. Looking at the `Verify` tab on the right, we see that there are four VCs in this function (each child of "Correctness" corresponds to a VC), and two of them are invalid. Selecting a VC highlights the corresponding code, in the `Code` pane, and displays the corresponding VC in the `Verification Condition` pane. Opening a VC node shows a list of steps on the corresponding basic path.



Let's take a look at the first failing VC. The corresponding basic path says that there is an element e at index i in the array a, and returns true. However, there is missing information: our basic path has no information about possible range of i. The postcondition, on the other hand, deals only with a specific range of the array. Thus, although the VC expresses the fact that an element was found, it doesn't express the fact that an element was found within the appropriate range. This makes the VC invalid. To fix this problem, we must include information about the range of i. We add the conjunct $i >= i$ to the loop annotation. The other bound, $i <= u$, is provided by the loop condition.

Let's take a look at the other failing VC. The corresponding basic path starts at the `for` loop, but doesn't enter it, and returns from the function. To make the VC for this basic path valid, we need to give πVC some information about what the loop is doing. As programmers, we know that loop is walking down the array, and if i is at a given point, it means that it has not found the element e at an index less than i. We can supply that information in the loop annotation, and try recompiling.



We did it! The green root node indicates that this program is completely verified. If this is a homework assignment, you would now be ready to submit.

Most programs will be more complicated than this, and will require multiple annotations, each with multiple conjuncts.

Submitting πVC Assignments

πVC submission is done electronically. You just click the `Submit` button above the code pane. You will receive an email confirmation of your submission.

Built-in Predicates

Predicates serve as syntactic shortcuts. For example, rather than writing

```
forall a,b. ((low <= a && a <= b && b <= high) -> arr[a]<=arr[b]),
```

you can just write

```
sorted(arr,low,high).
```

A full list of predicates supported by πVC is as follows.

```
predicate sorted(int[] arr, int low, int high) :=
  (forall a,b. ((low <= a && a <= b && b <= high) -> arr[a]<=arr[b]));

predicate partitioned(int[] arr, int low1, int high1, int low2, int high2) :=
  (forall a,b. ((low1 <= a && a <= high1 && low2 <= b && b <= high2)-> arr[a]<=arr[b]));

predicate eq(int[] arr1, int[] arr2, int low, int high) :=
  (forall a. ((low <= a && a <= high) -> arr1[a]=arr2[a]));
```

You can also define your own predicates by following the syntax used above. Just write them directly into a `.pi` file.

Bugs and Other Feedback

This is the first time that this version of πVC has been used in CS156. We don't anticipate any issues, but if you notice any bugs, or have any other feedback, then you can let us know using `Help->Report A Bug` or `Help->Give Feedback on PiVC`.

Tips on using πVC

Verifying a program requires a lot of thought. You will find it much more mentally taxing than a programming assignment. It is also very fun, and very satisfying when you finally prove a program correct.

Here are a few tips and strategies to help you out.

- One good strategy is to first start by supplying bounds on all the loop variables (e.g. `i`, `j`, and so forth). You usually need these, and you'll be sure not to forget them if you start with them. So for the loop `(i := 0; i < u; i := i + 1)`, the corresponding annotation should include the conjuncts `i>=0 && i<=u`. Why the $<=$ as opposed to $<$? Because at the end of the final iteration of the loop, just before it breaks out, it will be the case that $i = u$.
- Another strategy is to start by going through the program and annotating everything with what you as a programmer know it does. This way, you should have all the main loop assertions and can focus on providing the extra information needed to prove everything valid.
- The information contained under a specific VC node is an invaluable tool for figuring out why a VC is not valid. Walk through the path and the VC in your head and try to figure out why the logic doesn't hold and what you'd have to add to make it valid. Ask yourself the following question: given only the information contained in the annotation at the beginning of the basic path, and a list of steps in the basic path, can I conclude for sure that the annotation at the end of the path holds true? If the answer is "no", then add (to the beginning annotation) whatever extra information is needed.
- πVC doesn't automatically save programs when it compiles them, so remember to save things yourself.
- You can add your own assertions to a πVC program by using the syntax `@ assertion_goes_here;` on a line by itself. Sometimes this can be helpful if you want to see when something holds and when it doesn't. An assertion creates an additional basic path that begins at the preceding annotation, and ends at the assertion.
- Make sure you're familiar with the material in Chapters 5 and 6 of the textbook. Chapter 6 proposes some strategies to help you find inductive assertions, and also presents an extended example. You can also find more examples at the πVC website.
- Keep at it. Some programs and properties can be very hard to prove, but if you keep working at it you'll eventually manage it. Good luck!

Settings Menu

You can leave all settings on their defaults. However, if you are curious, here is what enabling each setting means:

- **Generate runtime assertions**
  For each array access `arr[i]`, an associated assertion `i>=0 && i<|arr|` is generated. For each division `x/y`, an associated assertion `y!=0` is generated. Although proving these things is not usually difficult, it's not something we require for the πVC assignments. Hence, this option is disabled by default.
- **Find inductive core**
  If a subset of your conjuncts proves the post-condition, yet one of your conjuncts is extraneous and breaks down the proof, πVC will detect the situation and notify you. This can make verification a little easier. Hence, this option is enabled by default.