Program Correctness: Mechanics

CS410
Fall 2019
What happens under the hood in Dafny?
Overview

Goal: specifying and proving properties of programs.

Model: Control Flow Graph, or the program itself.

Specifications: First Order Logic (FOL) formulas.

Proof Methods: Inductive Assertion Method, and Ranking Functions.
A Simple Language

5.1 pi: A Simple Imperative Language

@pre T
@post T

bool LinearSearch(int[] a, int ℓ, int u, int e) {
  for (int i := ℓ; i ≤ u; i := i + 1) {
    if (a[i] = e) return true;
  }
  return false;
}

Fig. 5.1. LinearSearch

guages: its data types do not include pointer or reference types; and it does not allow global variables, although it does have global constants (see Exercise 6.5). After reading this chapter and Chapter 12, the interested reader should consult the wide literature on program analysis to learn how the techniques of these chapters extend to reasoning about standard programming languages.

5.1.1 The Language

Because pi is superficially a C-like language with restrictions, we present the essential features of pi through examples.

Example 5.1. Figure 5.1 lists the function LinearSearch, which searches the range \([ℓ, u]\) of a range a of integers for a value e. It returns true if the given array contains the value between the lower bound ℓ and upper bound u. It behaves correctly only if 0 ≤ ℓ and u < |a|; otherwise, the array a is accessed outside of its domain \([0, |a| − 1]\).

Notice the lines beginning with @. They are program annotations, which we discuss in detail in the next section.

Example 5.2. Figure 5.2 lists the recursive function BinarySearch, which searches a range \([ℓ, u]\) of a sorted (weakly increasing: \(a[i] ≤ a[j]\) if \(i ≤ j\)) array a of integers for a value e. Like LinearSearch, it returns true if the given array contains the value between the lower bound ℓ and upper bound u. It behaves correctly only if 0 ≤ ℓ and u < |a|; otherwise, the array a is accessed outside of its domain \([0, |a| − 1]\).

One syntactic difference occurs in assignment, which is written := to distinguish it from the equality predicate =. We use = as the equality predicate, rather than ==, to correspond to the standard equality predicate of FOL. Finally, unlike C, pi has type bool and constants true and false.
Annotations

• An annotation is a First Order Logic formula $F$ whose free variables only include program variables.

• An annotation $F$ at program location $L$ means that $F$ holds whenever program control reaches $L$.

---

```plaintext
@pre T
@post T
bool LinearSearch(int[] a, int ℓ, int u, int e) {
    for @ T
        (int i := ℓ; i ≤ u; i := i + 1) {
            if (a[i] = e) return true;
        } @ i = u
    return false;
}
```
• **Precondition** indicates what is true upon entering the function. Free variables only include function parameters.

• **Postcondition**: indicates what is true upon exiting the function. Free variables only include function parameters and a special variable, `rv`, that refers to the return value.

```plaintext
@pre
@post
bool LinearSearch(int[] a, int ℓ, int u, int e) {
  for (int i := ℓ; i ≤ u; i := i + 1) {
    if (a[i] = e) return true;
  }
  return false;
}
```
Loop Invariant

- Loop Invariant holds at the beginning of each iteration.

\[
F \land \langle \text{condition} \rangle
\]

\[
F \land \neg \langle \text{condition} \rangle
\]

for

\[
\text{while } @ F
\]

\[
(\langle \text{initialize} \rangle; \langle \text{condition} \rangle; \langle \text{increment} \rangle) \} \}
\]

\[
\langle \text{body} \rangle
\]

\[
\text{while } @ F
\]

\[
(\langle \text{condition} \rangle) \} \}
\]

\[
\langle \text{body} \rangle
\]

\[
\langle \text{increment} \rangle
\]
Example

```java
@pre 0 \leq \ell \land u < |a|
@post rv \leftrightarrow \exists i. \ell \leq i \leq u \land a[i] = e

bool LinearSearch(int[] a, int \ell, int u, int e) {
    for
        @L:
            (int i := \ell; i \leq u; i := i + 1) {
                if (a[i] = e) return true;
            }
    return false;
}
```

Example 5.9. Figure 5.10 lists LinearSearch with a nontrivial loop invariant at L. It asserts that whenever control reaches L, the loop index is at least \ell and that a[j] \neq e for previously examined indices j. ■

Section 5.2 shows that loop invariants are crucial for constructing an inductive argument that a function obeys its specification.

Assertions In pi, one can add an annotation anywhere. When an annotation is not a function precondition, function postcondition, or loop invariant, we call it an assertion. Assertions allow programmers to provide a formal comment. For example, if at the statement i := i + k; the programmer thinks that k is positive, then the programmer can add an assertion stating that supposition:

```java
k > 0;
```

Later, the programmer's hypothesis about k is verified with formal verification at compile time or with dynamic assertion tests at runtime.

Runtime assertions are a special class of assertions. In most programming languages, runtime errors include division by 0, modulo by 0, and dereference of null. In particular, division by 0 and modulo by 0 cause hardware exceptions, while only some languages, such as Java, catch a dereference of null. In pi, runtime errors include division by 0, modulo by 0, and accessing an array out of bounds. The pi compiler generates runtime assertions to catch runtime errors.

Example 5.10. Figure 5.11 lists LinearSearch with runtime assertions. The array read a[i] is protected by the assertion that i is a legal index of a.

■
Assertions

We can add annotations anywhere in the program.

• **Assertions:** when they are not preconditions, postconditions, or loop invariants, they are simply called assertions.

![Code snippet](@ k > 0; i := i + k;)

Section 5.2 shows that loop invariants are crucial for constructing an inductive argument that a function obeys its specification.
Partial Correctness
Overview

A function is partially correct if when the function’s precondition is satisfied on entry, its postcondition is satisfied when it returns (if it ever does).
Some Definitions

- **Program States**: an assignment of values (of the proper type) to program variables.

\[ s: \{ pc \leftarrow L_1, l \leftarrow 1, u \leftarrow 3, i \leftarrow 3, a \leftarrow [4; 7; 1], rv \leftarrow [] \} \]

The state can also be represented by any logical formula in any theory.

```plaintext
@pre 0 \leq l \land u < |a|
@post rv \leftrightarrow \exists i. l \leq i \leq u \land a[i] = e
bool LinearSearch(int[] a, int l, int u, int e) {
    for
        @L: l \leq i \land (\forall j. l \leq j < i \rightarrow a[j] \neq e)
        (int i := l; i \leq u; i := i + 1) {
            if (a[i] = e) return true;
        }
    return false;
}
```
Partial Correctness

- Given pre/post conditions
  \[ F_{pre}, F_{post} \]
  \[ s_0[pc] = L_0 \]
  \[ s_0 \models F_{pre} \]

- The function may have both finite and infinite paths:
  \[ s_0 s_1 s_2 \ldots s_n \]
  \[ s_0 s_1 s_2 \ldots s_n \ldots \]

- The function is partially correct if for ever finite path:
  \[ s_0 \models F_{pre} \iff s_n \models F_{post} \]
How do we prove partial correctness?

How does Dafny work?
Overview

A function is **partially correct** if when the function's precondition is satisfied on entry, its postcondition is satisfied when it returns (if it ever does).

Can we do this by only using the precondition and postcondition?

- in most cases: NO!

We **prove** partial correctness of programs by the **Inductive Assertion Method**.

- For each function, we generate a **finite set of Verification Conditions (VC)**; if all VCs are correct, then the program is partially correct.
Some Definitions

- **Path**: sequence of program statements.
- **Basic Path**: a Path that starts at a precondition or a loop invariant, and ends at a loop invariant, an assertion, or a post condition.

```plaintext
@pre 0 ≤ ℓ ∧ u < |a|
@post rv ← ∃i. ℓ ≤ i ≤ u ∧ a[i] = e

bool LinearSearch(int[] a, int ℓ, int u, int e) {
    for
      @L: ℓ ≤ i ∧ (∀j. ℓ ≤ j < i → a[j] ≠ e)
      (int i := ℓ; i ≤ u; i := i + 1) {
        if (a[i] = e) return true;
      }
    return false;
}
```

- **@pre 0 ≤ ℓ ∧ u < |a|**: precondition
- **@post rv ← ∃i. ℓ ≤ i ≤ u ∧ a[i] = e**: postcondition
- **@L: ℓ ≤ i ∧ (∀j. ℓ ≤ j < i → a[j] ≠ e)**: loop guard
- **@post rv ← ∃j. ℓ ≤ j ≤ u ∧ a[j] = e**: assertion

Figures:
- **Fig. 5.16.** Basic paths of LinearSearch
- **Fig. 5.17.** Annotated version of LinearSearch
Inductive Assertion Method

- We reduce the reasoning about the function to reasoning about a finite set of basic paths.

- We reason about the basic paths, by reducing the reasoning to a Verification Condition (VC).

\[
\begin{align*}
\@ x & \geq 0 \\
x & := x + 1; \\
\@ x & \geq 1
\end{align*}
\]
P-Invariant vs. P-Inductive

• **P-Invariant**: an annotation $F$ at location $L$ of program $P$ is P-invariant iff whenever program reaches location $L$ during any computation with program state $s$, then $s \models F$.

  $$s[pc] = L \implies s \models F$$

• **P-Inductive**: if all verification conditions generated by the program are valid, then all program annotations are P-inductive.

**Theorem**: p-inductive implies p-invariant.

For iterative programs, finding an inductive annotation mostly amounts to discovery of an appropriate loop invariant.