What happens under the hood in Dafny?
The Calculus of Computation
Decision Procedures with Applications to Verification
Aaron R. Bradley
Zohar Manna
Springer
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

5.1.1 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 \([\ell, u]\) of an array \(a\) of integers for a value \(e\). It returns true if the given array contains the value between the lower bound \(\ell\) and upper bound \(u\). It behaves correctly only if \(0 \leq \ell\) and \(u < |a|\); otherwise, the array \(a\) is accessed outside of its domain \([0, |a| - 1]\). |\(a|\) denotes the length of array \(a\).

Observe that most of the syntax is similar to C. For example, the for loop sets \(i\) to be \(\ell\) initially and then executes the body of the loop and increments \(i\) by 1 as long as \(i \leq u\). Also, an integer array has type \(\text{int}[]\), which is constructed from base type \(\text{int}\). 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 \(\text{bool}\) and constants true and false.

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 \([\ell, u]\) of a sorted (weakly increasing: \(a[i] \leq a[j]\) if \(i \leq j\)) array \(a\) of integers for a value \(e\). Like LinearSearch, it returns true if the given array contains the value between the lower bound \(\ell\) and upper bound \(u\). It behaves correctly only if \(0 \leq \ell\) and \(u < |a|\); otherwise, the array \(a\) is accessed outside of its domain \([0, |a| - 1]\). |\(a|\) denotes the length of array \(a\).
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;
}
```
Example 5.5. In Example 5.1, we informally specified the behavior of `LinearSearch` as follows: `LinearSearch` returns true if the array `a` contains the value `e` in the range \([\ell, u]\). It behaves correctly only when \(\ell \geq 0\) and \(u < |a|\).

Function specifications formalize such statements. Figure 5.6 presents `LinearSearch` with its specification. The precondition asserts that the lower bound \(\ell\) should be at least 0 and that the upper bound \(u\) should be less than the length \(|a|\) of the array `a`. The postcondition asserts that the return value `rv` is true if `a[i] = e` for some index \(i \in [\ell, u]\) of `a`. ■

Example 5.6. A nontrivial precondition (a formula other than \(\top\)) is not always acceptable, especially if a function is public to a module. Figure 5.7 lists a more robust version of linear search. The formula \(0 \leq \ell \leq i \leq u < |a|\) abbreviates \(0 \leq \ell \land \ell \leq i \land i \leq u \land u < |a|\).

A nontrivial precondition is sometimes acceptable for a function that is private to a module. The verification method of this chapter checks that every instance of a call to such a function obeys the precondition. ■

- **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.

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

- Loop Invariant holds at the beginning of each iteration.

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

\[
\begin{align*}
\text{while} & \quad @ F \\
(\langle \text{condition} \rangle) & \{ \\
\langle \text{body} \rangle & \}
\end{align*}
\]

\[
\begin{align*}
\text{for} & \quad @ F \\
(\langle \text{initialize} \rangle ; \langle \text{condition} \rangle ; \langle \text{increment} \rangle) & \{ \\
\langle \text{body} \rangle & \}
\end{align*}
\]
Example

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

Watch the Dafny Video for “Find”
We can add annotations anywhere in the program.

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

How do we prove every program path satisfies the specification?

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.

```
(1)  (3)  (2), (4)
   @pre  L  @post
```

Example 5.14.

Figure 5.17 lists BubbleSort with loop invariants. The outer loop invariant at $L_1$ asserts that

- $i$ is in the range $[-1, |a| - 1]$ (if $|a| = 0$, then $i$ is initially $-1$);
- $a$ is sorted in the range $[i, |a| - 1]$;
- and $a$ is partitioned such that each element in the range $[0,i]$ is at most (less than or equal to) each element in the range $[i+1, |a| - 1]$.

Its inner loop invariant at $L_2$ asserts that

- $i$ is in the range $[1, |a| - 1]$, and $j$ is in the range $[0,i]$;
- $a$ is sorted in the range $[i, |a| - 1]$ as in the outer loop;
- $a$ is partitioned as in the outer loop;
- and $a$ is also partitioned such that each element in the range $[0,j-1]$ is at most $a[j]$.
```
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.

---

```c
@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

(1)

@post

(2), (4)

@L: ℓ ≤ i ∧ (∀j. ℓ ≤ j < i → a[j] ≠ e)

@L: ℓ ≤ i ∧ (∀j. ℓ ≤ j < i → a[j] ≠ e)

assume i > u;
rv := false;
@post rv ↔ ∃j. ℓ ≤ j ≤ u ∧ a[j] = e
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 discovering an appropriate loop invariant.
Watch all Dafny Videos on iterative examples:

Find, Quotient/Remainder, Strengthening, Robot, Partition
Total Correctness
Total Correctness

- To prove termination of functions, we use well-founded relations.

- Ranking functions are a convenient way of dealing with well-founded relations.

- We need to prove that annotations are inductive.
- We need to prove that the ranking function decreases along each basic path, beginning and ending with ranking functions.
Watch the Dafny Video on termination
What about function calls?
Basic Paths: Functions

- A function's post condition summarizes the effect of calling the function, by relating its return value to its parameters.

- Replacing function calls by their summaries makes listing of basic paths and the reasoning about the function local.

```
@pre 0 \leq \ell \land u < |a| \land \text{sorted}(a, \ell, u)
@post rv \iff \exists i. \ell \leq i \leq u \land a[i] = e

bool BinarySearch(int[] a, int \ell, int u, int e) {
    if (\ell > u) return false;
    else {
        int m := (\ell + u) \div 2;
        if (a[m] = e) return true;
        else if (a[m] < e) return BinarySearch(a, m + 1, u, e);
        else return BinarySearch(a, \ell, m - 1, e);
    }
}
```
Basic Paths: Functions

- A function's post condition summarizes the effect of calling the function, by relating its return value to its parameters.

- Replacing function calls by their summaries makes listing of basic paths and the reasoning about the function local.

```java
@pre 0 ≤ ℓ ∧ u < |a| ∧ sorted(a, ℓ, u)
@post rv ← ∃i. ℓ ≤ i ≤ u ∧ a[i] = e
bool BinarySearch(int[] a, int ℓ, int u, int e) {
  if (ℓ > u) return false;
  else {
    int m := (ℓ + u) div 2;
    if (a[m] = e) return true;
    else if (a[m] < e) {
      @R₁: 0 ≤ m + 1 ∧ u < |a| ∧ sorted(a, m + 1, u);
      return BinarySearch(a, m + 1, u, e);
    } else {
      @R₂: 0 ≤ ℓ ∧ m − 1 < |a| ∧ sorted(a, ℓ, m − 1);
      return BinarySearch(a, ℓ, m − 1, e);
    }
  }
}
```
Function Summaries

• A functions post condition summarizes the effect of calling the function, by relating its return value to its parameters.

• An appropriate function summary is inductive (same as P-inductive).

• To construct a proof, an inductive function summary is required.
Watch the recursive Robot Dafny video to understand the difference between inductive and non-inductive summaries!
Read the full binary search example from the recommended book chapter!
Your Input

vs

Theorem Prover's Help
Motivation for Strategies

Main Challenge: discovering the extra information to make the method succeed: loop invariants, ...

We know how to reduce the checking of an annotated function to a finite set of basic paths.

We can use the SMT technology to automatically check the validity of these paths.

You think and strategize to come up with these annotations in the first place.
Challenges

- Writing **function specification** (pre/post-conditions) requires **human ingenuity**.
- Simple and generic assertions, such as ruling out runtime errors, can be generated **automatically**.
- Writing **loop invariants** also requires **human ingenuity**.
- Writing **inductive loop invariants** are specifically **hard**.

- A lot of research has been done for this.
- Example: linear and polynomial relations between variables can be discovered.
Next: Hoare Logic