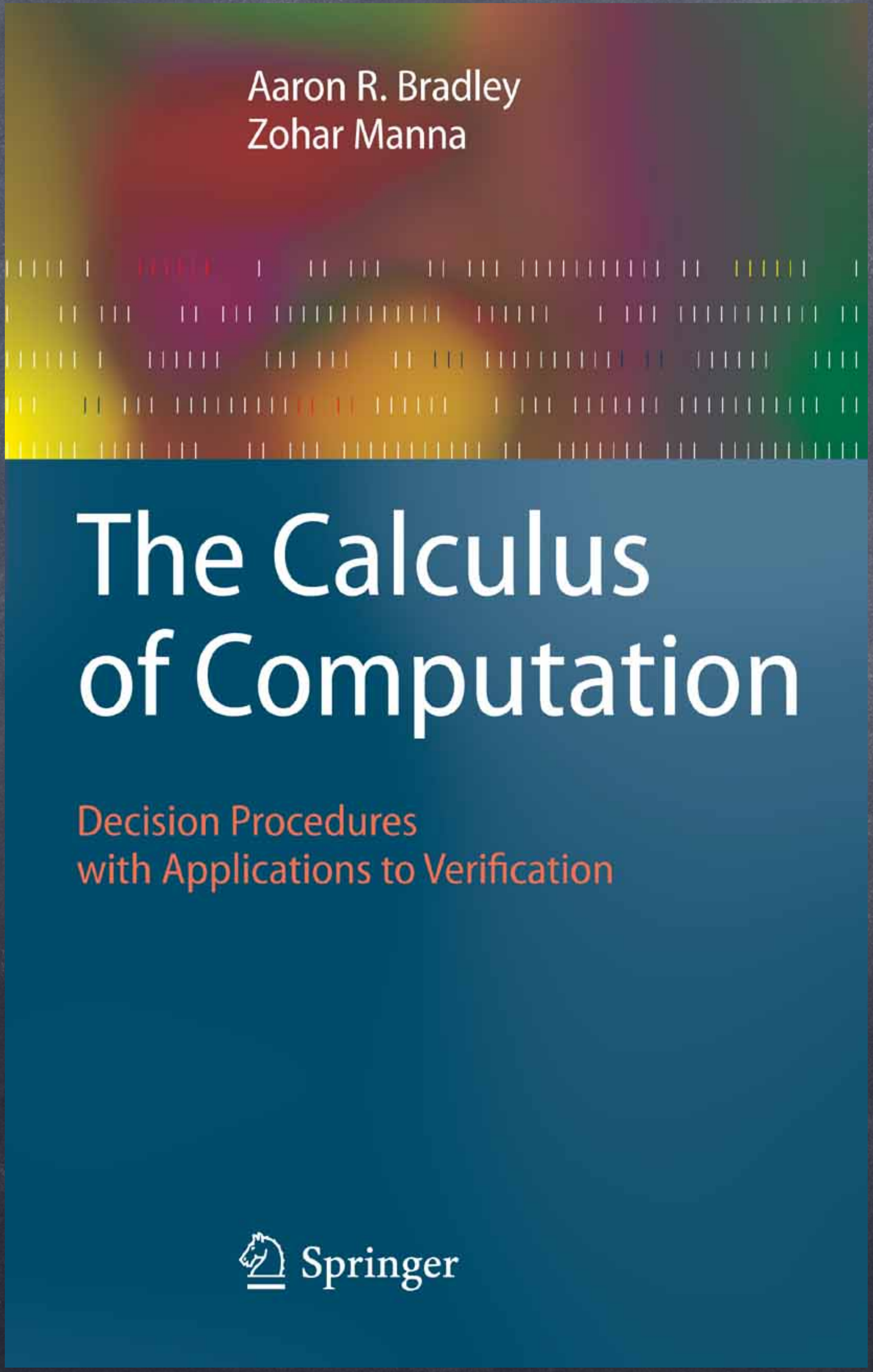
Program Correctness: Mechanics

CS410

Fall 2020

What happens under the hood in Dafny?

Reference



Aaron R. Bradley
Zohar Manna
The Calculus of Computation
Decision Procedures with Applications to Verification
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

Annotations

```
@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;
}
```

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.

```
@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;
  } @ 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.

@pre

@post

bool LinearSearch(int[] a, int ℓ, 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.

while
  @ F
  $(\langle condition \rangle)$ {
  $\langle body \rangle$
}

$F \wedge \langle condition \rangle$

$F \wedge \neg \langle condition \rangle$

$\langle initialize \rangle$;
while
  @ F
  $(\langle condition \rangle)$ {
  $\langle body \rangle$
  $\langle increment \rangle$
}

for
  @ F
  $(\langle initialize \rangle;\ \langle condition \rangle;\ \langle increment \rangle)$ {
  $\langle body \rangle$
}

Example

```
@pre 0 ≤ ℓ ∧ u < |a|
@post rv ↔ ∃i. ℓ ≤ i ≤ u
bool LinearSearch(int[] ... int u, int e) {
  for
    @L :
    (int i ... ; i := i + 1) {
    if ( ... return true;
  }
  ... false;
```

Watch the Dafny Video for "Find"

Assertions

We can add annotations anywhere in the program.

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

$$@\ k > 0;$$
$$i := i + k;$$

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.

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:

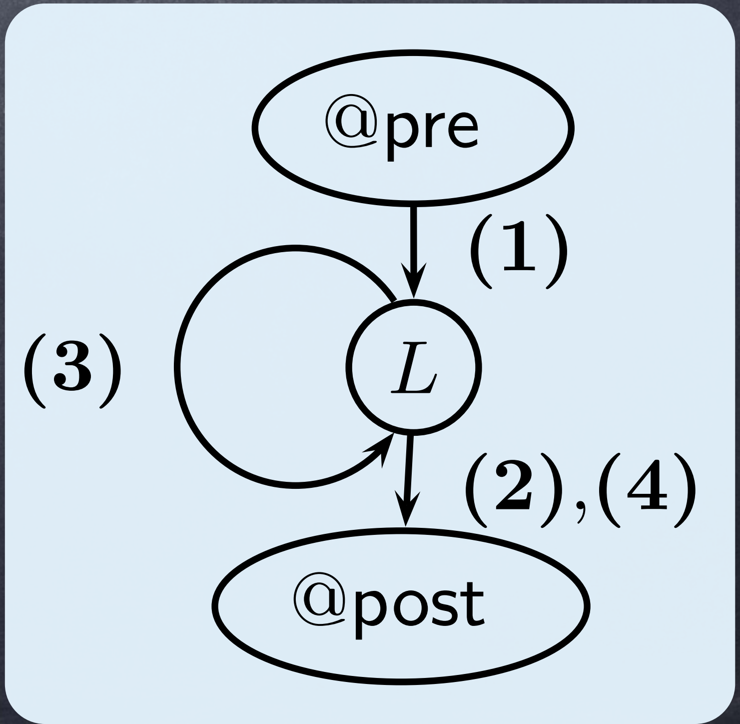
$$s_0 \models F_{pre} \implies s_n \models F_{post}$$

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.

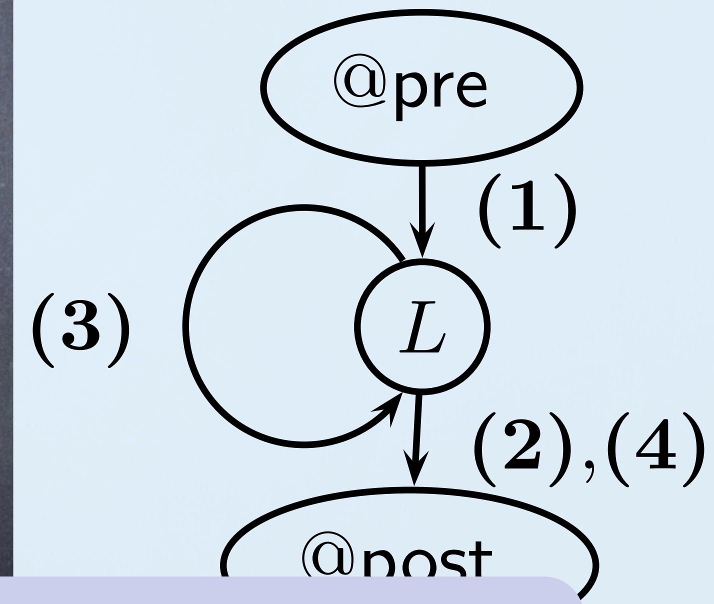


Some Definitions

- 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.

$@\text{pre } 0 \le \ell \ \wedge \ u < |a|$
$@\text{post } rv \ \leftrightarrow \ \exists i.\ \ell \le i \le u \ \wedge \ a[i] = e$
$\texttt{bool}$ LinearSearch$(\texttt{int}[]\ a,\ \texttt{int}\ \ell,\ \texttt{int}\ u,\ \texttt{int}\ e)\ \{$
  $\texttt{for}$
    $@L:\ \ell \le i \ \wedge \ (\forall j.\ \ell \le j < i \ \rightarrow \ a[j] \ne e)$
    $(\texttt{int}\ i := \ell;\ i \le u;\ i := i + 1)\ \{$
    $\texttt{if}\ (a[i] = e)\ \texttt{return true};$
  $\}$
  $\texttt{return false};$
$\}$



$@L:\ \ell \le i \ \wedge \ (\forall j.\ \ell \le j < i \ \rightarrow \ a[j] \ne e)$
$@L:\ \ell \le i \ \wedge \ (\forall j.\ \ell \le j < i \ \rightarrow \ a[j] \ne e)$
$\texttt{assume}\ i > u;$
$rv := \texttt{false};$
$@\text{post } rv \ \leftrightarrow \ \exists j.\ \ell \le j \le u \ \wedge \ a[j] = e$

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).

$$@\ x \geq 0$$
$$x := x + 1;$$
$$@\ x \geq 1$$

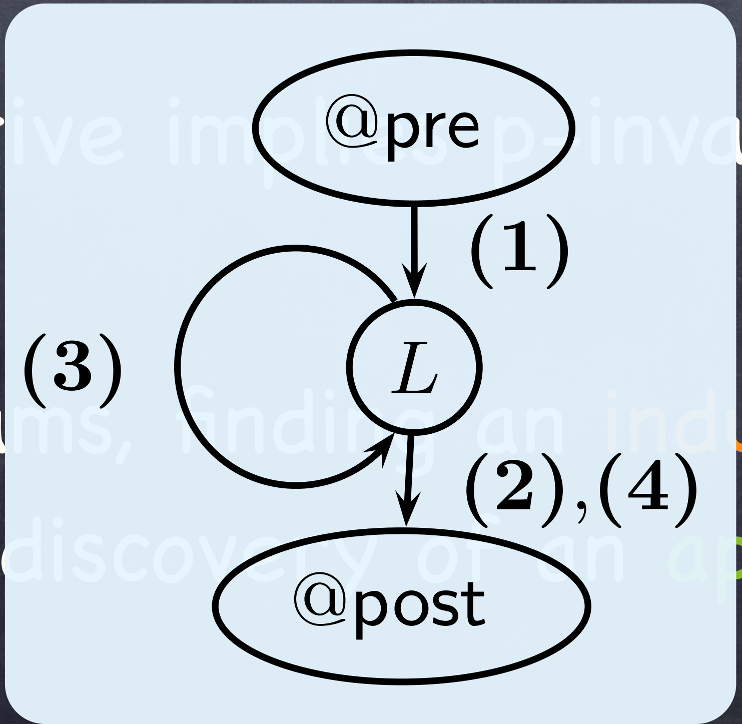
P-Invariant vs. P-Inductive

- 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 |= 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.

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 functions 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.

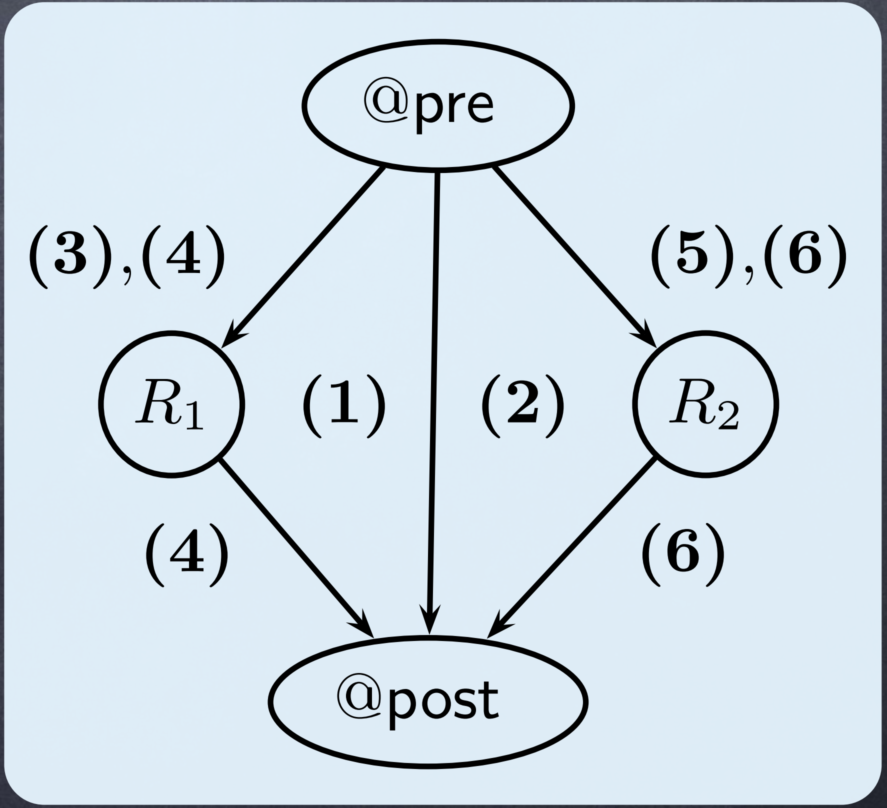
$$\text{@pre } 0 \leq \ell \;\wedge\; u < |a| \;\wedge\; \text{sorted}(a, \ell, u)$$
$$\text{@post } rv \;\leftrightarrow\; \exists i.\ \ell \leq i \leq u \;\wedge\; a[i] = e$$

```
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) return BinarySearch(a, m + 1, u, e);
    else return BinarySearch(a, ℓ, m − 1, e);
  }
}
```

Basic Paths: Functions

- A functions 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 ≤ 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 run time 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