Program Correctness: Mechanics

CS410
Fall 2019
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.
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.

```cpp
@pre 0 \leq \ell \land u < |a| \land \text{sorted}(a, \ell, u)
@post rv \leftrightarrow \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);
    }
}
```

Example 5.7. Figure 5.8 lists BinarySearch with its specification. As expected, its postcondition is identical to the postcondition of LinearSearch. However, its precondition also states that the array `a` is sorted. The sorted predicate is defined in the combined theory of integers and arrays, $T_Z \cup T_A$:

```
\text{sorted}(a, \ell, u) \iff \forall i, j. \ell \leq i \leq j \leq u \implies a[i] \leq a[j].
```
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.

```plaintext
@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);
        }
    }
}
```
as other assertions, such as runtime assertions. So far, we have the following

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
</table>
| (1)  | @pre $0 \leq \ell \land u < |a| \land \text{sorted}(a, \ell, u)$  
assume $\ell > u$;  
rv := false;  
@post rv $\leftrightarrow \exists i. \ell \leq i \leq u \land a[i] = e$ |
| (2)  | @pre $0 \leq \ell \land u < |a| \land \text{sorted}(a, \ell, u)$  
assume $\ell \leq u$;  
m := $(\ell + u) \div 2$;  
assume $a[m] = e$;  
rv := true;  
@post rv $\leftrightarrow \exists i. \ell \leq i \leq u \land a[i] = e$ |
| (3)  | @pre $0 \leq \ell \land u < |a| \land \text{sorted}(a, \ell, u)$  
assume $\ell \leq u$;  
m := $(\ell + u) \div 2$;  
assume $a[m] \neq e$;  
assume $a[m] < e$;  
@R$_1$: $0 \leq m + 1 \land u < |a| \land \text{sorted}(a, m + 1, u)$ |
| (4)  | @pre $0 \leq \ell \land u < |a| \land \text{sorted}(a, \ell, u)$  
assume $\ell \leq u$;  
m := $(\ell + u) \div 2$;  
assume $a[m] \neq e$;  
assume $a[m] \geq e$;  
assume $v_1 \leftarrow \exists i. m + 1 \leq i \leq u \land a[i] = e$;  
rv := $v_1$;  
@post rv $\leftrightarrow \exists i. \ell \leq i \leq u \land a[i] = e$ |
| (5)  | @pre $0 \leq \ell \land u < |a| \land \text{sorted}(a, \ell, u)$  
assume $\ell \leq u$;  
m := $(\ell + u) \div 2$;  
assume $a[m] \neq e$;  
assume $a[m] \geq e$;  
@R$_2$: $0 \leq \ell \land m - 1 < |a| \land \text{sorted}(a, \ell, m - 1)$ |
| (6)  | @pre $0 \leq \ell \land u < |a| \land \text{sorted}(a, \ell, u)$  
assume $\ell \leq u$;  
m := $(\ell + u) \div 2$;  
assume $a[m] \neq e$;  
assume $a[m] < e$;  
assume $v_2 \leftarrow \exists i. \ell \leq i \leq m - 1 \land a[i] = e$;  
rv := $v_2$;  
@post rv $\leftrightarrow \exists i. \ell \leq i \leq u \land a[i] = e$ |
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.
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.