Program Correctness: Mechanics

CS410
Fall 2017
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.
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) return BinarySearch(a, m + 1, u, e);
    else return BinarySearch(a, ℓ, 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.

```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) {
            @R_1: 0 ≤ m + 1 ∧ u < |a| ∧ sorted(a, m + 1, u);
            return BinarySearch(a, m + 1, u, e);
        }
        else {
            @R_2: 0 ≤ ℓ ∧ m - 1 < |a| ∧ sorted(a, ℓ, m - 1);
            return BinarySearch(a, ℓ, 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.

```cpp
@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) {
            @R_1: 0 ≤ m + 1 ∧ u < |a| ∧ sorted(a, m + 1, u);
            return BinarySearch(a, m + 1, u, e);
        } else {
            @R_2: 0 ≤ ℓ ∧ m − 1 < |a| ∧ sorted(a, ℓ, m − 1);
            return BinarySearch(a, ℓ, 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.

```
@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);
        }
    }
}
```
The first function call is
@pre $0 \leq l \land u < |a| \land \text{sorted}(a, l, u)$
assume $l > u$;
rv := false;
@post rv $\leftrightarrow \exists i. l \leq i \leq u \land a[i] = e$

The second function call is
@pre $0 \leq l \land u < |a| \land \text{sorted}(a, l, u)$
assume $l \leq u$;
m := $(l + u) \div 2$;
assume $a[m] \neq e$;
assume $a[m] < e$;
asume $v_1$ $\leftrightarrow \exists i. m + 1 \leq i \leq u \land a[i] = e$;
rv := $v_1$;
@post rv $\leftrightarrow \exists i. l \leq i \leq u \land a[i] = e$

The penultimate lines of

Next, given that the precondition holds (from path

Because

Suppose that

where

These assertions are treated in the same way

Visualization of basic paths of

The first function call is
@pre $0 \leq l \land u < |a| \land \text{sorted}(a, l, u)$
assume $l > u$;
r := false;
@post rv $\leftrightarrow \exists i. l \leq i \leq u \land a[i] = e$

The second function call is
@pre $0 \leq l \land u < |a| \land \text{sorted}(a, l, u)$
assume $l \leq u$;
m := $(l + u) \div 2$;
assume $a[m] \neq e$;
assume $a[m] < e$;
asume $v_1$ $\leftrightarrow \exists i. m + 1 \leq i \leq u \land a[i] = e$;
rv := $v_1$;
@post rv $\leftrightarrow \exists i. l \leq i \leq u \land a[i] = e$
Function Summaries

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

- A function’s 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).
Function Summaries

• A function's **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**.
Total Correctness
To prove termination of functions, we use well-founded relations.

```c
int[] BubbleSort(int[] a0) {
    int[] a := a0;
    for @ T
        (int i := |a| − 1; i > 0; i := i − 1) {
            for @ T
                (int j := 0; j < i; j := j + 1) {
                    @ 0 ≤ j < |a| ∧ 0 ≤ j + 1 < |a|
                    if (a[j] > a[j + 1]) {
                        int t := a[j];
                        a[j] := a[j + 1];
                        a[j + 1] := t;
                    }
                }
        }
    return a;
}
```
Total Correctness

- To prove termination of functions, we use well-founded relations.
- Ranking functions are a convenient way of dealing with well-founded relations.

```plaintext
int[] BubbleSort(int[] a0) {
    int[] a := a0;
    for @ T
        (int i := |a| - 1; i > 0; i := i - 1) {
            for @ T
                (int j := 0; j < i; j := j + 1) {
                    @ 0 ≤ j < |a| ∧ 0 ≤ j + 1 < |a|;
                    if (a[j] > a[j + 1]) {
                        int t := a[j];
                        a[j] := a[j + 1];
                        a[j + 1] := t;
                    }
                }
        }
    return a;
}
```
To prove termination of functions, we use well-founded relations.

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

```c
int[] BubbleSort(int[] a0) {
    int[] a := a0;
    for
        @L1 : i + 1 ≥ 0
        ↓ (i + 1, i + 1)
        (int i := |a| - 1; i > 0; i := i - 1) {
            for
                @L2 : i + 1 ≥ 0 ∧ i − j ≥ 0
                ↓ (i + 1, i − j)
                (int j := 0; j < i; j := j + 1) {
                    if (a[j] > a[j + 1]) {
                        int t := a[j];
                        a[j] := a[j + 1];
                        a[j + 1] := t;
                    }
                }
        }
    return a;
}
```
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.
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.

@\(L_1\) : \(i + 1 \geq 0\)
\(\downarrow L_1 : (i + 1, i + 1)\)
assume \(i > 0\);
\(j := 0;\)
\(\downarrow L_2 : (i + 1, i - j)\)

@\(L_2\) : \(i + 1 \geq 0 \land i - j \geq 0\)
\(\downarrow L_2 : (i + 1, i - j)\)
assume \(j < i;\)
assume \(a[j] > a[j + 1];\)
\(t := a[j];\)
\(a[j] := a[j + 1];\)
\(a[j + 1] := t;\)
\(j := j + 1;\)
\(\downarrow L_2 : (i + 1, i - j)\)

@\(L_2\) : \(i + 1 \geq 0 \land i - j \geq 0\)
\(\downarrow L_2 : (i + 1, i - j)\)
assume \(j \geq i;\)
\(i := i - 1;\)
\(\downarrow L_1 : (i + 1, i + 1)\)
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.

@\(L_1\): \(i + 1 \geq 0\)

\[\downarrow L_1: (i + 1, i + 1)\]

assume \(i > 0\);

\(j := 0;\)

\[\downarrow L_2: (i + 1, i - j)\]

@\(L_2\): \(i + 1 \geq 0 \land i - j \geq 0\)

\[\downarrow L_2: (i + 1, i - j)\]

assume \(j < i;\)

assume \(a[j] > a[j + 1];\)

\(t := a[j];\)

\(a[j] := a[j + 1];\)

\(t[j + 1] := t;\)

\(j := j + 1;\)

\[\downarrow L_2: (i + 1, i - j)\]
bubbleSort(int[] a) {
    int[] a := a;
    for @L1: i + 1 ≥ 0 ↓L1: (i + 1, i + 1) assume i > 0;
        j := 0;
        ↓L2: (i + 1, i − j) assume a[j] > a[j + 1];
        t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
        j := j + 1;
    }
    return a;
}

Fig. 5.23. BubbleSort with annotations to prove termination

method. We leave this step to the reader. It only remains to prove that the
functions decrease along each basic path.

The relevant basic paths are the following:

1. @L1: i + 1 ≥ 0 ↓L1: (i + 1, i + 1)
   assume i > 0;
   j := 0;
   ↓L2: (i + 1, i − j)

2. @L2: i + 1 ≥ 0 ∧ i − j ≥ 0 ↓L2: (i + 1, i − j)
   assume j < i;
   assume a[j] > a[j + 1];
   t := a[j];
   a[j] := a[j + 1];
   a[j + 1] := t;
   j := j + 1;
↓L2: (i + 1, i − j)
@L_1: \( i + 1 \geq 0 \)
\[ L_1: (i + 1, i + 1) \]
assume \( i > 0 \);
\( j := 0; \)
\[ L_2: (i + 1, i - j) \]
\[ i + 1 \geq 0 \land i > 0 \rightarrow (i + 1, i - 0) <_2 (i + 1, i + 1) \]
We leave this step to the reader. It only remains to prove that the statement

\[ i + 1 \geq 0 \land i > 0 \rightarrow (i + 1, i - 0) \prec_2 (i + 1, i + 1) \]

induces a verification condition. So there is nothing to prove. The exiting path leads to the termination argument. The entering path does not begin with a ranking function.

The relevant basic paths are the following:

1. For termination purposes, paths:
   - \(@L_1: i + 1 \geq 0\)
   - \(\downarrow L_1: (i + 1, i + 1)\)
   - **assume** \(i > 0;\)
   - \(j := 0;\)
   - \(\downarrow L_2: (i + 1, i - j)\)

2. \(@L_2: i + 1 \geq 0 \land i - j \geq 0\)
   - \(\downarrow L_2: (i + 1, i - j)\)
   - **assume** \(j < i;\)
   - **assume** \(a[j] \leq a[j + 1];\)
   - \(j := j + 1;\)
   - \(\downarrow L_2: (i + 1, i - j)\)
This renaming process preserves the value of \( \text{halts} \). Pathfore, we rename the variables of 

\[
\begin{align*}
\@L_1 &: \ i + 1 \geq 0 \\
\downarrow \text{L}_1 &: \ (i + 1, i + 1) \\
\text{assume} \ i > 0; \\
\text{j} &: = 0; \\
\downarrow \text{L}_2 &: \ (i + 1, i - j)
\end{align*}
\]

\( i + 1 \geq 0 \land i > 0 \rightarrow (i + 1, i - 0) <_2 (i + 1, i + 1) \)

\[
\begin{align*}
\@L_2 &: \ i + 1 \geq 0 \land i - j \geq 0 \\
\downarrow \text{L}_2 &: \ (i + 1, i - j) \\
\text{assume} \ j < i; \\
\text{assume} \ a[j] \leq a[j + 1]; \\
\text{j} &: = j + 1; \\
\downarrow \text{L}_2 &: \ (i + 1, i - j)
\end{align*}
\]

\[
\begin{align*}
i + 1 \geq 0 \land i - j \geq 0 \land j < i \rightarrow (i + 1, i - (j + 1)) <_2 (i + 1, i - j)
\end{align*}
\]
Program Correctness: Strategies
Motivation for Strategies

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

What you don’t know is how to come up with these annotations in the first place.
Challenges

Writing **function specification** (pre/post-conditions) requires **human ingenuity**.
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**.
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**.
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**.
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.
First Step: Basic Facts

- Basic facts are **obvious** facts such as loop index ranges.
- Include basic facts in loop invariants.
First Step: Basic Facts

Basic facts are obvious facts such as loop index ranges.

Include basic facts in loop invariants.

Example 6.1.

Consider the loop of `LinearSearch` (see also Figure 5.1):

```plaintext
for @L : T
    (int i := l; i ≤ u; i := i + 1) {
        if (a[i] = e) return true;
    }
```
First Step: Basic Facts

- **Basic facts** are obvious facts such as loop index ranges.
- Include basic facts in loop invariants.

```plaintext
for @L: ℓ ≤ i ≤ u + 1
    (int i := ℓ; i ≤ u; i := i + 1) {
        if (a[i] = e) return true;
    }
```
Example: bubble sort

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, \( T_{\mathbb{Z}} \cup T_{\mathbb{A}} \):

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

Example 5.8. Figure 5.9 lists `BubbleSort` with its specification. Given any array, the returned array is sorted. Of course, other properties are desirable.

```plaintext
@pre T
@post sorted(rv, 0, |rv| - 1)
int[] BubbleSort(int[] a0) {
    int[] a := a0;
    for @ T
        (int i := |a| - 1; i > 0; i := i - 1) {
            for @ T
                (int j := 0; j < i; j := j + 1) {
                    if (a[j] > a[j + 1]) {
                        int t := a[j];
                        a[j] := a[j + 1];
                        a[j + 1] := t;
                    }
                }
        }
    return a;
}
```
Bubble sort: basic facts

```plaintext
for
  @L_1: -1 \leq i < |a|
  (int i := |a| - 1; i > 0; i := i - 1) {
    for
      @L_2: 0 < i < |a| \land 0 \leq j \leq i
        (int j := 0; j < i; j := j + 1) {
          if (a[j] > a[j + 1]) {
            int t := a[j];
            a[j] := a[j + 1];
            a[j + 1] := t;
            }
        }
  }
```

Note that the loops modify just the elements of $a$, not $a$ itself. Therefore, we could add the annotation $|a| = |a_0|$ to both loops. Such an annotation would be useful if the postcondition asserted, for example, that $|rv| = |a_0|$. For the property that we address (sorted($rv$, $0$, $|rv| - 1$)), this annotation is not useful.

6.1.2 The Precondition Method

Basic facts provide a foundation for more interesting information. The precondition method (also called the “backward substitution” or “backward propagation” method) is a strategy for developing more interesting information in a structured way. Again, we emphasize that the method is a heuristic, not an algorithm: it provides some guidance for the human rather than replacing the human’s intuition and ingenuity.

The precondition method consists of the following steps:

1. Identify a fact $F$ that is known at one location $L$ in the function (@$L$: $F$) but that is not supported by annotations earlier in the function.
2. Repeat:
   a) Compute the weakest preconditions of $F$ backward through the function, ending at loop invariants or at the beginning of the function.
   b) At each new annotation location $L'$, generalize the facts on the new formula $F'(L')$.

We illustrate the technique through examples.
Weakest Precondition
The Precondition Method is a strategy to find more interesting information out of basic facts in a structured way.
The Precondition Method

- The Precondition Method is a strategy to find more interesting information out of basic facts in a structured way.

- Also called, backward propagation or backward substitution.
Weakest Preconditions

- Weakest Precondition is a predicate transformer:

\[ wp : FOL \times Stmt \rightarrow FOL \]

such that for a state \( t \), a program statement \( S \), and an annotation \( F \), if:

\[ t \models wp(F, S) \]

and if \( S \) is executed in state \( t \) to generate state \( t' \), then:

\[ t' \models F \]
Weakest Preconditions

- Weakest Precondition is a predicate transformer:

\[ wp : FOL \times Stmt \rightarrow FOL \]

such that for a state \( t \), a program statement \( S \), and an annotation \( F \), if:

\[ t \models wp(F, S) \]

and if \( S \) is executed in state \( t \) to generate state \( t' \), then:

\[ t' \models F \]

This situation is visualized in Figure 12.1 (a). The region labeled \( F \) is the set of states that satisfy \( F \); similarly, the region labeled \( wp(F, S) \) is the set of
• **Weakest Precondition** is defined (constructively):

  - **assumptions**: What must hold before assume $c$, so that $F$ can hold afterwards?

  - **assignments**: What must hold before $v := e$, so that $F(v)$ can hold afterwards?

  - **sequence of statements**: How do we combine the weakest preconditions in a basic path?
• **Weakest Precondition** is defined (constructively):

  • **assumptions**: What must hold before `assume c`, so that `F` can hold afterwards?

  \[ wp(F, \text{assume } c) \iff c \rightarrow F \]

  • **assignments**: What must hold before `v:=e`, so that `F(v)` can hold afterwards?

  • **sequence of statements**: How do we combine the weakest preconditions in a basic path?
• **Weakest Precondition** is defined (constructively):

  - **assumptions**: What must hold before assume $c$, so that $F$ can hold afterwards?

    \[
    wp(F, \text{assume } c) \iff c \rightarrow F
    \]

  - **assignments**: What must hold before $v := e$, so that $F(v)$ can hold afterwards?

    \[
    wp(F[v], v := e) \iff F[e]
    \]

  - **sequence of statements**: How do we combine the weakest preconditions in a basic path?
• **Weakest Precondition** is defined (constructively):

- **assumptions**: What must hold before `assume c`, so that `F` can hold afterwards?

\[
wp(F, assume \ c) \iff c \rightarrow F
\]

- **assignments**: What must hold before `v:=e`, so that `F(v)` can hold afterwards?

\[
wp(F[v], v := e) \iff F[e]
\]

- **sequence of statements**: How do we combine the weakest preconditions in a basic path?

\[
wp(F, S_1; \ldots; S_n) \iff wp(wp(F, S_n), S_1; \ldots; S_{n-1})
\]
Incidentally ...

**Verification Condition** of a basic path:

\[ F \rightarrow \text{wp}(G, S_1; \ldots; S_n) \]
Incidentally ...

- Verification Condition of a basic path:

  
  \[ F \rightarrow wp(G, S_1; \ldots; S_n) \]

  ...

  \begin{align*}
  @ x &\geq 0 \\
  x &:= x + 1; \\
  @ x &\geq 1
  \end{align*}

  ...

  \begin{align*}
  @ F \\
  S_1; \\
  \vdots \\
  S_n; \\
  @ G
  \end{align*}
Incidentally ...

- **Verification Condition** of a basic path:

  
  \[ F \rightarrow wp(G, S_1; \ldots; S_n) \]

\[
\{ x \geq 0 \} x := x + 1 \{ x \geq 1 \} : x \geq 0 \rightarrow \text{wp}(x \geq 1, x := x + 1)
\]
Example

\[ @L : F : \ell \leq i \land (\forall j. \ell \leq j < i \rightarrow a[j] \neq e) \]

\[ S_1 : \text{assume } i \leq u; \]

\[ S_2 : \text{assume } a[i] = e; \]

\[ S_3 : rv := \text{true}; \]

\[ @\text{post } G : rv \leftrightarrow \exists j. \ell \leq j \leq u \land a[j] = e \]

\[ \text{wp}(G, S_1; S_2; S_3) \]

\[ \iff \text{wp(wp}(rv \leftrightarrow \exists j. \ell \leq j \leq u \land a[j] = e, rv := \text{true}), S_1; S_2) \]

\[ \iff \text{wp}(\text{true} \leftrightarrow \exists j. \ell \leq j \leq u \land a[j] = e, S_1; S_2) \]

\[ \iff \text{wp}(\exists j. \ell \leq j \leq u \land a[j] = e, S_1; S_2) \]

\[ \iff \text{wp}(\exists j. \ell \leq j \leq u \land a[j] = e, \text{assume } a[i] = e), S_1) \]

\[ \iff \text{wp}(a[i] = e \rightarrow \exists j. \ell \leq j \leq u \land a[j] = e, S_1) \]

\[ \iff \text{wp}(a[i] = e \rightarrow \exists j. \ell \leq j \leq u \land a[j] = e, \text{assume } i \leq u) \]

\[ \iff i \leq u \rightarrow (a[i] = e \rightarrow \exists j. \ell \leq j \leq u \land a[j] = e) \]
The Precondition Method is a strategy to find more interesting information out of basic facts in a structured way.

Also called, backward propagation or backward substitution.
The Precondition Method

The Precondition Method is a strategy to find more interesting information out of basic facts in a structured way.

Also called, backward propagation or backward substitution.

(1) Identify a fact F that is known at location L, but is not supported by earlier annotations.
The Precondition Method

The Precondition Method is a strategy to find more interesting information out of basic facts in a structured way. Also called, backward propagation or backward substitution.

1. Identify a fact F that is known at location L, but is not supported by earlier annotations.

2. Repeat:
The Precondition Method

*The Precondition Method* is a strategy to find more interesting information out of basic facts in a structured way.

Also called, *backward propagation* or *backward substitution*.

1. Identify a fact $F$ that is known at location $L$, but is not supported by earlier annotations.

2. Repeat:
   
   (a) Compute the *weakest precondition* of $F$ backward, ending at loop invariants or the beginning.
The Precondition Method is a strategy to find more interesting information out of basic facts in a structured way. Also called, backward propagation or backward substitution.

(1) Identify a fact $F$ that is known at location $L$, but is not supported by earlier annotations.

(2) Repeat:

(a) Compute the weakest precondition of $F$ backward, ending at loop invariants or the beginning.

(b) At each new annotation location $L'$, generalize the new fact to new formula $F'$. 
Example

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

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

@L : F_1 : \ell \leq i \leq u + 1
S_1 : assume i > u;
S_2 : rv := false;
@post F_2 : rv \leftrightarrow \exists j. \ell \leq j \leq u \land a[j] = e
Example

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

VC: \[ \ell \leq i \leq u + 1 \land i > u \rightarrow \neg (\exists j. \ell \leq j \leq u \land a[j] = e) \]

@L: \( F_1: \ell \leq i \leq u + 1 \)

\( S_1: \) assume \( i > u; \)

\( S_2: \) \( rv := \) false;

@post \( F_2: \) \( rv \leftrightarrow \exists j. \ell \leq j \leq u \land a[j] = e \)
Consider basic path of Example 5.13 but with the current loop invariant LinearSearch.

\[ \begin{align*}
\text{for} & \quad \forall \ell \leq i \leq u + 1 \\
& \quad \text{(int } i := \ell; \ i \leq u; \ i := i + 1) \{ \\
& \quad \text{if } (a[i] = e) \ \text{return true;} \\
\} \\
& \quad \text{return false;}
\end{align*} \]

\[ \begin{align*}
\@L : & \quad F_1 : \ell \leq i \leq u + 1 \\
& \quad S_1 : \ \text{assume } i > u; \\
& \quad S_2 : \ rv := \text{false}; \\
& \quad @\text{post } F_2 : \ rv \leftrightarrow \exists j. \ \ell \leq j \leq u \land a[j] = e
\end{align*} \]

**VC:** \[ \ell \leq i \leq u + 1 \land i > u \rightarrow \neg(\exists j. \ \ell \leq j \leq u \land a[j] = e) \]

**F₂** is a fact not supported by earlier annotations.
Example 6.3. Consider the loop of \textit{LinearSearch} (see also Figure 5.1), annotated with basic facts:

\[
\begin{align*}
&\text{for } 1 \leq i \leq u + 1 \\
&\quad \text{(int } i := \ell; i \leq u; i := i + 1 ) \\
&\text{if } (a[i] = e) \text{ return true} \\
&\text{return false}
\end{align*}
\]

The postcondition of \textit{LinearSearch} is

\[
rv \leftrightarrow \exists i. \ell \leq i \leq u \land a[i] = e.
\]

Consider basic path (4) of Example 5.13 but with the current loop invariant substituted for the first assertion:

(4) 
\[
\begin{align*}
&\text{@L: } F_1: \ell \leq i \leq u + 1 \\
&S_1: \text{assume } i > u \\
&S_2: rv := \text{false} \\
&\text{@post } F_2: rv \leftrightarrow \exists j. \ell \leq j \leq u \land a[j] = e
\end{align*}
\]

Note that we continue to number basic paths as they were numbered in Example 5.13. The VC \( \{F_1\} S_1; S_2 \{F_2\} \) is not \((T_Z \cup T_A)\)-valid. Essentially, the antecedent does not assert anything useful about the content of \(a\).

Write the consequence \(F\): \(\forall j. \ell \leq j \leq u \rightarrow a[j] \neq e\) by pushing in the negation.

\(F\) says that if \textit{LinearSearch} exits via \(S_2\), then no element of \(a\) in the range \([\ell, u]\) is \(e\). But \(F\) is not supported by the current loop invariant at \(L\). In short, \(F_2\) is a fact that is not supported by earlier annotations.

Having identified an unsupported fact, we compute preconditions. To propagate \(F_2\) back to the loop invariant, compute \(\text{wp}(F_2, S_1; S_2) \equiv \text{wp}(\text{wp}(F_2, rv := \text{false}), S_1) \equiv \text{wp}(F_2 \{rv \mapsto \text{false}\}, S_1) \equiv \text{wp}(F_2 \{rv \mapsto \text{false}\}, \text{assume } i > u) \equiv i > u \rightarrow F_2 \{rv \mapsto \text{false}\} \equiv i > u \rightarrow \forall j. \ell \leq j \leq u \rightarrow a[j] \neq e\)
Example 6.3. Consider the loop of LinearSearch (see also Figure 5.1), annotated with basic facts:

\[
\ell \leq i \leq u + 1
\]

\[
\begin{align*}
\text{for } & i := \ell; & i \leq u; & i := i + 1 \\
\text{if } & (a[i] = e) & \text{return true} \\
\text{return false}
\end{align*}
\]

The postcondition of LinearSearch is

\[
rv \leftrightarrow \exists i. \ell \leq i \leq u \land a[i] = e.
\]

Consider basic path (4) of Example 5.13 but with the current loop invariant substituted for the first assertion:

\[
\begin{align*}
\text{(4) } & \ell \leq i \leq u + 1 \\
\text{S}_1 & : \text{assume } i > u \\
\text{S}_2 & : rv := \text{false} \\
\text{F}_2 & : rv \leftrightarrow \exists j. \ell \leq j \leq u \land a[j] = e
\end{align*}
\]

Note that we continue to number basic paths as they were numbered in Example 5.13. The VC

\[
\{ F_1 \} S_1; S_2 \{ F_2 \}
\]

\[
\ell \leq i \leq u + 1 \land i > u \rightarrow \neg (\exists j. \ell \leq j \leq u \land a[j] = e)
\]

is not \((T \cup TA)\)-valid. Essentially, the antecedent does not assert anything useful about the content of \(a\). Write the consequent \(F\):

\[
\forall j. \ell \leq j \leq u \rightarrow a[j] \neq e
\]

by pushing in the negation. \(F\) says that if LinearSearch exits via \(S_2\), then no element of \(a\) in the range \([\ell, u]\) is \(e\). But \(F\) is not supported by the current loop invariant at \(L\). In short, \(F_2\) is a fact that is not supported by earlier annotations.

Having identified an unsupported fact, we compute preconditions. To propagate \(F_2\) back to the loop invariant, compute \(wp(F_2, S_1; S_2)\)

\[
\iff wp(wp(F_2, rv := \text{false}), S_1)
\]

\[
\iff wp(F_2\{rv \rightarrow \text{false}\}, S_1)
\]

\[
\iff wp(F_2\{rv \rightarrow \text{false}\}, \text{assume } i > u)
\]

\[
\iff i > u \rightarrow F_2\{rv \rightarrow \text{false}\}
\]

\[
\iff i > u \rightarrow \forall j. \ell \leq j \leq u \rightarrow a[j] \neq e
\]

Guessing from the last line that this generalization should work:
Guessing from the last line that this \textit{generalization} should work:

\[
\forall j. \, \ell \leq j < i \rightarrow a[j] \neq e
\]
wp(\(F_2, S_1; S_2\))
\[\iff wp(wp(F_2, rv := \text{false}), S_1)\]
\[\iff wp(F_2\{rv \mapsto \text{false}\}, S_1)\]
\[\iff wp(F_2\{rv \mapsto \text{false}\}, \text{assume } i > u)\]
\[\iff i > u \rightarrow F_2\{rv \mapsto \text{false}\}\]
\[\iff i > u \rightarrow \forall j. \; l \leq j \leq u \rightarrow a[j] \neq e\]

Guessing from the last line that this **generalization** should work:

\[\forall j. \; l \leq j < i \rightarrow a[j] \neq e\]

How do we increase our confidence? Try another path.
wp(F_2, S_1; S_2)
⇔ wp(wp(F_2, rv := false), S_1)
⇔ wp(F_2\{rv \mapsto false\}, S_1)
⇔ wp(F_2\{rv \mapsto false\}, assume i > u)
⇔ i > u \rightarrow F_2\{rv \mapsto false\}
⇔ i > u \rightarrow \forall j. \ell \leq j \leq u \rightarrow a[j] \neq e

Guessing from the last line that this generalization should work:
\[
\forall j. \ell \leq j < i \rightarrow a[j] \neq e
\]

How do we increase our confidence? Try another path.

@L: H: ?
S_1: assume i \leq u;
S_2: assume a[i] \neq e;
S_3: i := i + 1;
@L: G: i > u \rightarrow \forall j. \ell \leq j \leq u \rightarrow a[j] \neq e
\[ wp(G, S_1; S_2; S_3) \]
\[ \equiv wp(wp(G, i := i + 1), S_1; S_2) \]
\[ \equiv wp(G\{i := i + 1\}, S_1; S_2) \]
\[ \equiv wp(wp(G\{i := i + 1\}, assume a[i] \neq e), S_1) \]
\[ \equiv wp(a[i] \neq e \rightarrow G\{i := i + 1\}, S_1) \]
\[ \equiv wp(a[i] \neq e \rightarrow G\{i := i + 1\}, assume i \leq u) \]
\[ \equiv i \leq u \rightarrow a[i] \neq e \rightarrow G\{i := i + 1\} \]
\[ \equiv i \leq u \land a[i] \neq e \land i + 1 > u \rightarrow \forall j. l \leq j \leq u \rightarrow a[j] \neq e \]
\[ \equiv i = u \land a[u] \neq e \rightarrow \forall j. l \leq j \leq u \rightarrow a[j] \neq e \]
\[ \equiv i = u \land a[u] \neq e \rightarrow \forall j. l \leq j \leq u - 1 \rightarrow a[j] \neq e \]
\[ \equiv i = u \land a[u] \neq e \rightarrow \forall j. l \leq j \leq i - 1 \rightarrow a[j] \neq e \]
We compute one backward iteration through the loop to increase our confidence in particular the antecedent already asserts that

So, we can settle on this:

\[ \forall j. \ell \leq j < i \rightarrow a[j] \neq e \]

and, add it to the loop invariant.
\[ \text{wp}(G, S_1; S_2; S_3) \]  
\[ \iff \text{wp}(\text{wp}(G, i := i + 1), S_1; S_2) \]  
\[ \iff \text{wp}(G\{i := i + 1\}, S_1; S_2) \]  
\[ \iff \text{wp}(\text{wp}(G\{i := i + 1\}, \text{assume } a[i] \neq e), S_1) \]  
\[ \iff \text{wp}(a[i] \neq e \rightarrow G\{i := i + 1\}, S_1) \]  
\[ \iff \text{wp}(a[i] \neq e \rightarrow G\{i := i + 1\}, \text{assume } i \leq u) \]  
\[ \iff i \leq u \rightarrow a[i] \neq e \rightarrow G\{i := i + 1\} \]  
\[ \iff i \leq u \land a[i] \neq e \land i + 1 > u \rightarrow \forall j. \ l \leq j \leq u \rightarrow a[j] \neq e \]  
\[ \iff i = u \land a[u] \neq e \rightarrow \forall j. \ l \leq j \leq u \rightarrow a[j] \neq e \]  
\[ \iff i = u \land a[u] \neq e \rightarrow \forall j. \ l \leq j \leq u - 1 \rightarrow a[j] \neq e \]  
\[ \iff i = u \land a[u] \neq e \rightarrow \forall j. \ l \leq j \leq i - 1 \rightarrow a[j] \neq e \]  

So, we can settle on this:  
\[ \forall j. \ l \leq j < i \rightarrow a[j] \neq e \]  

and, add it to the loop invariant.

**General Trick:** replace fixed terms (bounds, indices, etc) with terms that evolve according to the loop counter.
Example: Bubble sort

```plaintext
for
  @L_1: -1 ≤ i < |a|
  (int i := |a| - 1; i > 0; i := i - 1) {
    for
      @L_2: 0 < i < |a| ∧ 0 ≤ j ≤ i
      (int j := 0; j < i; j := j + 1) {
        if (a[j] > a[j + 1]) {
          int t := a[j];
          a[j] := a[j + 1];
          a[j + 1] := t;
        }
      }
  }
return a;
```
Example: Bubble sort

```java
for
    @L1: -1 ≤ i < |a|
    (int i := |a| − 1; i > 0; i := i − 1) {
        for
            @L2: 0 < i < |a| ∨ 0 ≤ j ≤ i
            (int j := 0; j < i; j := j + 1) {
                if (a[j] > a[j + 1]) {
                    int t := a[j];
                    a[j] := a[j + 1];
                    a[j + 1] := t;
                }
            }
    }
return a;
```

\[ F : \text{sorted}(rv, 0, |rv| − 1) \]
@L_{1}: G: ?
S_1: assume i \leq 0;
S_2: rv := a;
@post F: sorted(rv, 0, |rv| - 1)
@L_1: \ G: ?
S_1: \text{assume } i \leq 0;
S_2: \ rv := a;
@post F: \text{sorted}(rv, 0, |rv| - 1)

wp(F, S_1; S_2)
@L_1: \ G \ : \ ? \\
S_1 : \ assume \ i \leq 0; \\
S_2 : \ rv := a; \\
@post \ F : \ sorted(rv, 0, |rv| - 1) \\

wp(F, S_1; S_2) \\
F' : \ i \leq 0 \ \rightarrow \ \ sorted(a, 0, |a| - 1)
Recalling the trick, we generalize to:

\[ @L_1 : \; G : ? \]
\[ S_1 : \; \text{assume } i \leq 0; \]
\[ S_2 : \; rv := a; \]
\[ @post \; F : \; \text{sorted}(rv, 0, |rv| - 1) \]

\[ \text{wp}(F, S_1; S_2) \]
\[ F' : i \leq 0 \rightarrow \text{sorted}(a, 0, |a| - 1) \]
\[ G : \; \text{sorted}(a, i, |a| - 1) \]
Recalling the trick, we generalize to:

And, have the new invariant:

\[ @L_1 : \quad -1 \leq i < |a| \land \text{sorted}(a, i, |a| - 1) \]
Recalling the trick, we generalize to:

And, have the new invariant:

Continue by propagating to the inner loop:
Recalling the trick, we generalize to:

And, have the new invariant:

Continue by propagating to the inner loop:

\texttt{@L}_1: \quad \texttt{G} : ? \\
\texttt{S}_1: \quad \texttt{assume } i \leq 0; \\
\texttt{S}_2: \quad \texttt{rv} := a; \\
\texttt{@post } \texttt{F} : \quad \texttt{sorted}(rv, 0, \vert rv \vert - 1) \\
\texttt{wp}(F, S_1; S_2) \\
\texttt{F'}: \quad i \leq 0 \rightarrow \texttt{sorted}(a, 0, \vert a \vert - 1) \\
\texttt{G}: \quad \texttt{sorted}(a, i, \vert a \vert - 1) \\
\texttt{@L}_1: \quad \texttt{-1 \leq i < \vert a \vert \land sorted(a, i, \vert a \vert - 1)} \\
\texttt{@L}_2: \quad \texttt{H} : ? \\
\texttt{S}_1: \quad \texttt{assume } j \geq i; \\
\texttt{S}_2: \quad i := i - 1; \\
\texttt{@L}_1: \quad \texttt{G} : \texttt{sorted}(a, i, \vert a \vert - 1)
Recalling the trick, we generalize to:

\[ \text{wp}(F, S_1; S_2) \]

And, have the new invariant:

\[ @L_1 : -1 \leq i < |a| \land \text{sorted}(a, i, |a| - 1) \]

Continue by propagating to the inner loop:

\[ @L_2 : H : ? \]

\[ S_1 : \text{assume } j \geq i; \]

\[ S_2 : i := i - 1; \]

\[ @L_1 : G : \text{sorted}(a, i, |a| - 1) \]
Recalling the trick, we generalize to: 

And, have the new invariant:

Continue by propagating to the inner loop:
Recalling the trick, we generalize to:

And, have the new invariant:

Continue by propagating to the inner loop:

Too strong??
Let’s give it a try:

@L_1: G: \text{sorted}(a, i, |a| - 1)
S_1: \text{assume } i > 0;
S_2: j := 0;
@L_2: H'': \text{sorted}(a, i - 1, |a| - 1)
Let's give it a try:

@L_1: G: sorted(a, i, |a| - 1)
S_1: assume i > 0;
S_2: j := 0;
@L_2: H'': sorted(a, i - 1, |a| - 1)

G → wp(H'', assume i > 0; j := 0)
Let's give it a try:

@L_1: \( G : \text{sorted}(a, i, |a| - 1) \)
\( S_1: \text{assume } i > 0; \)
\( S_2: j := 0; \)
@L_2: \( H'' : \text{sorted}(a, i - 1, |a| - 1) \)

\[ G \to \text{wp}(H'', \text{assume } i > 0; j := 0) \]

\[ \text{sorted}(a, i, |a| - 1) \land i > 0 \to \text{sorted}(a, i - 1, |a| - 1) \]
Let's give it a try:

\@L_1: \ G: \text{sorted}(a, i, |a| - 1) \\
S_1: \text{assume } i > 0; \\
S_2: \ j := 0; \\
\@L_2: \ H'': \text{sorted}(a, i - 1, |a| - 1)

\[ G \rightarrow \text{wp}(H'', \text{assume } i > 0; \ j := 0) \]

\[ \text{sorted}(a, i, |a| - 1) \land i > 0 \rightarrow \text{sorted}(a, i - 1, |a| - 1) \]

This is not a valid formula! Let's try a weaker version:
Let's give it a try:

@L_1: G: \text{sorted}(a, i, \lvert a \rvert - 1)
\begin{align*}
S_1 & : \text{assume } i > 0; \\
S_2 & : j := 0; \\
@L_2: H'' & : \text{sorted}(a, i - 1, \lvert a \rvert - 1)
\end{align*}

\text{sorted}(a, i, \lvert a \rvert - 1) \land i > 0 \rightarrow \text{sorted}(a, i - 1, \lvert a \rvert - 1)

This is not a valid formula! Let's try a weaker version:

\begin{align*}
H & : \text{sorted}(a, i, \lvert a \rvert - 1)
\end{align*}
So far ...

```plaintext
for
  @L_1: \ -1 \leq i < |a| \land \text{sorted}(a, i, |a| - 1)
  (\text{int } i := |a| - 1; \ i > 0; \ i := i - 1) \{
  for
    @L_2: \ 0 < i < |a| \land 0 \leq j \leq i \land \text{sorted}(a, i, |a| - 1)
    (\text{int } j := 0; \ j < i; \ j := j + 1) \{
      if (a[j] > a[j + 1]) \{
        \text{int } t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      \}
    \}
  \}
\}
```
A Strategy

Proofs in general require insights beyond generalizing formulas obtained through precondition method.
A Strategy

Proofs in general require insights beyond generalizing formulas obtained through precondition method.

Strategy: decompose the specification into atomic properties and analyze each separately.
A Strategy

Proofs in general require insights beyond generalizing formulas obtained through precondition method.

Strategy: decompose the specification into atomic properties and analyze each separately.

(1) Assert basic facts.
A Strategy

Proofs in general require insights beyond generalizing formulas obtained through precondition method.

Strategy: decompose the specification into atomic properties and analyze each separately.

(1) Assert basic facts.

(2) Repeat:
A Strategy

Proofs in general require insights beyond generalizing formulas obtained through precondition method.

Strategy: decompose the specification into atomic properties and analyze each separately.

(1) Assert basic facts.

(2) Repeat:

(a) Use the precondition method to propagate annotations.
A Strategy

Proofs in general require insights beyond generalizing formulas obtained through precondition method.

Strategy: decompose the specification into atomic properties and analyze each separately.

(1) Assert basic facts.

(2) Repeat:

(a) Use the precondition method to propagate annotations.

(b) Formalize an insight.
Back to our example

for
  @L_1: -1 \leq i < |a| \land \text{sorted}(a, i, |a| - 1)
  (\text{int } i := |a| - 1; \ i > 0; \ i := i - 1) \{ 
  for
    @L_2: 0 < i < |a| \land 0 \leq j \leq i \land \text{sorted}(a, i, |a| - 1)
    (\text{int } j := 0; \ j < i; \ j := j + 1) \{ 
      if (a[j] > a[j + 1]) \{ 
        \text{int } t := a[j];
        a[j] := a[j + 1];
        a[j + 1] := t;
      \}
    \}
  \}
Back to our example

\begin{verbatim}
for
  @L_1: -1 \leq i < |a| \land \text{sorted}(a, i, |a| - 1)
  (int i := |a| - 1; i > 0; i := i - 1) {
    for
      @L_2: 0 < i < |a| \land 0 \leq j \leq i \land \text{sorted}(a, i, |a| - 1)
      (int j := 0; j < i; j := j + 1) {
        if (a[j] > a[j + 1]) {
          int t := a[j];
          a[j] := a[j + 1];
          a[j + 1] := t;
        }
      }
  }
\end{verbatim}

At every iteration, all values in the range \([0, j - 1]\) are at most \(a[j]\).
Back to our example

for
  @L_1: \ -1 \leq i < |a| \land \text{sorted}(a, i, |a| - 1)
  (\text{int } i := |a| - 1; \ i > 0; \ i := i - 1) \{ \\
  for
    @L_2: \ 0 < i < |a| \land 0 \leq j \leq i \land \text{sorted}(a, i, |a| - 1)
    (\text{int } j := 0; \ j < i; \ j := j + 1) \{ \\
      if (a[j] > a[j + 1]) \{ \\
        \text{int } t := a[j]; \\
        a[j] := a[j + 1]; \\
        a[j + 1] := t; \\
      \}
    \}
  \}

At every iteration, all values in the range [0, j \ - 1] are at most a[j].

F: \ partitioned(a, 0, j \ - 1, j, j)
Example 6.6. We resume our analysis of BubbleSort from Example 6.5. Some cogitation (and observation of sample traces; see Figure 5.4) suggests that BubbleSort exhibits the following behavior: the inner loop propagates the largest value of the unsorted region to the right side of the unsorted region, thus expanding the sorted region. At every iteration, \( j \) is the index of the largest value found so far. In other words, all values in the range \([0, j - 1]\) are at most \( a[j]\):

\[
F : \text{partitioned}(a, 0, j - 1, j, j)
\]

This observation should be added as an annotation at \( L_2 \). Having gained new insight into BubbleSort, we return to the precondition method and propagate \( F \) back to the outer loop at \( L_1 \) along the path \( (2) \):

\[
\begin{align*}
@L_1 & : H : \? \\
S_1 & : \text{assume } i > 0; \\
S_2 & : j := 0; \\
@L_2 & : F : \text{partitioned}(a, 0, j - 1, j, j)
\end{align*}
\]
Thus expanding the sorted region. At every iteration, largest value of the unsorted region to the right side of the unsorted region, \( F \) at most largest value found so far. In other words, all values in the range \([0,1]\) are moved the largest element of the unsorted region to the sorted region. In other words, the sorted range \([0,j)\) must contain the largest elements of \([j+1,n]\).

Example 6.6. This observation should be added as an annotation at cogitation (and observation of sample traces; see Figure 5.4) suggests that some further meditation enlightens us with the following: the sorted region exhibits the following behavior: the inner loop propagates the partitioned \([0,j)\) along the path \( i > 0 \nrightarrow partitioned(a,0,−1,0,0) \).

---

@\(L_1\) : \( H : ? \)
\( S_1 \) : assume \( i > 0 \);
\( S_2 \) : \( j := 0 \);
@\(L_2\) : \( F \) : partitioned\((a,0,j−1,j,j)\)
insight into largest value of the unsorted region to the right side of the unsorted region, \( F \), at most largest value found so far. In other words, all values in the range \([0, F]\) are moved the largest element of the unsorted region to the sorted region. In every iteration, \( i > 0 \) → \( \text{partitioned}(a, 0, -1, 0, 0) \)

Trivially valid, so it does not contribute any new information.

```plaintext
@L_1 : H : ?
S_1 : assume \( i > 0 \);
S_2 : j := 0;
@L_2 : F : \text{partitioned}(a, 0, j - 1, j, j)
```

for
   \[ @L_1 : \quad -1 \leq i < |a| \land \text{sorted}(a, i, |a| - 1) \]
   \[(\text{int } i := |a| - 1; \ i > 0; \ i := i - 1) \} \]
   for
      \[ @L_2 : \quad \left[ 0 < i < |a| \land 0 \leq j \leq i \right. \]
      \[ \land \text{partitioned}(a, 0, j - 1, j, j) \land \text{sorted}(a, i, |a| - 1) \]
      \[(\text{int } j := 0; \ j < i; \ j := j + 1) \} \]
      if \((a[j] > a[j + 1])\) \{ \]
         \[ \text{int } t := a[j]; \]
         \[ a[j] := a[j + 1]; \]
         \[ a[j + 1] := t; \]
      \}\]
Thus expanding the sorted region. At every iteration, at most the largest value found so far. In other words, all values in the range \([0, \text{\#} - 1]\). The result is trivially valid according to the definition of \(\text{partitioned}\).

Some further meditation enlightens us with the following: the sorted region \([\text{\#} - 1, \#]\) contains the largest elements of \(\text{\#} - 1\) partitioned \([0, \text{\#} - 1, \text{\#}, \text{\#}, \text{\#}, \text{\#}]\). As in Example 6.5. Some formula is possible but not ideal. Instead, consider the path from \(\text{\#} - 1\) to the inner loop was unsuccessful. Similarly, \(\text{\#} - 0\) was a candidate to annotate the inner loop such that the \(\text{\#} - 1\) to \(\text{\#} - 0\) could be annotated. The annotations are not yet inductive.

\[
\begin{align*}
@L_1 & : H : ? \\
S_1 & : \text{assume } i > 0; \\
S_2 & : j := 0; \\
@L_2 & : F : \text{partitioned}(a, 0, j - 1, j, j)
\end{align*}
\]

Trivially valid, so it does not contribute any new information.

\[
\begin{align*}
\text{for} \\
@L_1 : -1 \leq i < |a| \land \text{sorted}(a, i, |a| - 1) \\
(\text{int } i := |a| - 1; i > 0; i := i - 1) \{ \\
\text{for} \\
@L_2 : \left[\begin{array}{c}
0 < i < |a| \land 0 \leq j \leq i \\
\land \text{partitioned}(a, 0, j - 1, j, j) \land \text{sorted}(a, i, |a| - 1) \\
\end{array}\right] \\
(\text{int } j := 0; j < i; j := j + 1) \{ \\
\text{if } (a[j] > a[j + 1]) \{ \\
\text{int } t := a[j]; \\
a[j] := a[j + 1]; \\
a[j + 1] := t;
\}
\}
\}
\]

\(G : \text{partitioned}(a, 0, i, i + 1, |a| - 1)\)
Trivially valid, so it does not contribute any new information.

```
for
  @L1: -1 ≤ i < |a| ∧ sorted(a, i, |a| − 1)
  (int i := |a| − 1; i > 0; i := i − 1) {
  for
    @L2: [0 < i < |a| ∧ 0 ≤ j ≤ i
      ∧ partitioned(a, 0, j − 1, j, j) ∧ sorted(a, i, |a| − 1)]
    (int j := 0; j < i; j := j + 1) {
    if (a[j] > a[j + 1]) {
      int t := a[j];
      a[j] := a[j + 1];
      a[j + 1] := t;
    }
  }
}
```

The sorted region must contain the largest element of the unsorted region.
@L_2 : H : ?
assume \( j \geq i \);
i := i - 1;
@L_1 : G : \text{partitioned}(a, 0, i, i + 1, |a| - 1)
This observation should be added as an annotation at VC for the path is valid. In other words, seek a formula that is supported by the annotation such formula is.

Fails to generalize to:

\[
\text{partitioned}(a, 0, i - 1, i, |a| - 1)
\]
In this section, we bring together the concepts studied in this and the last chapters through a single example. We prove that these new annotations result in Figure 5.17.

Recall from Example 6.5 that in the propagation of QuickSort, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.

QuickSort is a wrapper to an always halts and assigns $j$. As in Example 6.5, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.

Recall from Example 6.5 that in the propagation of QuickSort, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.

Recall from Example 6.5 that in the propagation of QuickSort, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.

Recall from Example 6.5 that in the propagation of QuickSort, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.

Recall from Example 6.5 that in the propagation of QuickSort, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.

Recall from Example 6.5 that in the propagation of QuickSort, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.

Recall from Example 6.5 that in the propagation of QuickSort, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.

Recall from Example 6.5 that in the propagation of QuickSort, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.

Recall from Example 6.5 that in the propagation of QuickSort, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.

Recall from Example 6.5 that in the propagation of QuickSort, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.

Recall from Example 6.5 that in the propagation of QuickSort, the first line of the outer loop assigns $i$ to the inner loop was unsuccessful. Similarly, this observation should be added as an annotation at $\ell$, which sorts array $a$, $i$.
Find the strongest formula $H$ annotating the inner loop that is supported by the annotation $G$ of the outer loop.

@L_2: \[ H : ? \]
assume \( j \geq i \);
\( i := i - 1 \);
@L_1: \[ G : \text{partitioned}(a, 0, i, i + 1, |a| - 1) \]

@L_1: \[ G : \text{partitioned}(a, 0, i, i + 1, |a| - 1) \]
S_1: assume \( i > 0 \);
S_2: \( j := 0 \);
@L_2: \[ H : ? \]

Fails to generalize to:

\[ \text{partitioned}(a, 0, i - 1, i, |a| - 1) \]
Find the strongest formula $H$ annotating the inner loop that is supported by the annotation $G$ of the outer loop.

\[
\text{partitioned}(a, 0, i, i + 1, |a| - 1) \rightarrow \text{wp}(H, S_1; S_2)
\]
int[] BubbleSort(int[] a0) {
    int[] a := a0;
    for (@L1: \[ -1 \leq i < \mid a \mid \]
                            \[ \land \text{partitioned}(a, 0, i, i + 1, \mid a \mid - 1) \]
                            \[ \land \text{sorted}(a, i, \mid a \mid - 1) \]
        (int i := \mid a \mid - 1; i > 0; i := i - 1) {
            for (@L2: \[ 1 \leq i < \mid a \mid \land 0 \leq j \leq i \]
                                \[ \land \text{partitioned}(a, 0, i, i + 1, \mid a \mid - 1) \]
                                \[ \land \text{partitioned}(a, 0, j - 1, j, j) \]
                                \[ \land \text{sorted}(a, i, \mid a \mid - 1) \]
            (int j := 0; j < i; j := j + 1) {
                if (a[j] > a[j + 1]) {
                    int t := a[j];
                    a[j] := a[j + 1];
                    a[j + 1] := t;
                }
            }
        }
    return a;
}