Automated Program Verification and Testing
15414/15614 Fall 2016
Lecture 16: Proving Termination, Inductive Annotations

Matt Fredrikson
mfredrik@cs.cmu.edu

October 27, 2016
Today’s Lecture

- Continue discussing termination
- Strategies for developing good annotations
We need to prove that the program always terminates

Intuitively, we’ll do the following:

1. Find a set $S$ with a certain kind of “finite” ordering

2. Find a function $\delta$ that maps program states to $S$

3. Prove that $\delta$ always decreases as the program executes

$\delta$ is called a **ranking function**

**Idea:** If program diverges, $\delta$ would decrease infinitely
Let $\prec$ be a binary predicate over some set $S$

Well-Founded Relations

$\prec$ is a **well-founded relation** iff there is no infinite sequence

$$s_1, s_2, s_3, \ldots \text{ in } S$$

where each element is *less than* its predecessor:

$$s_1 \succ s_2 \succ s_3 \succ \cdots$$

In other words, each decreasing sequence in $S$ is finite.
Lexicographic Relations

Given sets $S_1, \ldots, S_n$ and relations $\prec_1, \ldots, \prec_m$, let $S = S_1 \times \cdots \times S_n$

Then $(s_1, \ldots, s_n) \prec (t_1, \ldots, t_n)$ iff:

- At some position $i$, $s_i \prec_i t_i$
- and for all preceding positions $j$, $s_j = t_j$

Work from left to right, comparing each pair in turn

Stop when a left element is $\prec$ a right one

For example,

$$(1, 2, 3, 4, 5) <_5 (1, 2, 4, 3, 5)$$
Annotating Ranking Functions

We annotate ranking functions in code with ↓

Needed in function prototype, head of loops

For example:

- ↓ i, as long as i’s domain has a well-founded ordering
- ↓ u − l + 1, assuming u, l are naturals
- ↓ (i + 1, j − 1), same assumption as above

Basic paths now have ranking functions at beginning and end

- Usually accompanied by assertion at the beginning
- By themselves at the end, to prove the path decreases
We produce the verification condition:

\[
P \rightarrow \text{wlp}(c_1; \cdots ; c_n, \delta(x) < \delta(x'))[x/x']
\]

- The value of \(\delta(x)\) at the end is less than at the beginning
- Don’t want wlp to modify value of \(x\) at the beginning
- Rename it to \(x'\), then subst. \(x\) back in after computing wlp
Let's compute the VC:

\[
i \geq 1 \rightarrow \text{wlp}(i := i - 1, i < i')[i/i']
\]

becomes

\[
i \geq 1 \rightarrow (i - 1 < i')[i/i']
\]

becomes

\[
i \geq 1 \rightarrow i - 1 < i
\]

This is valid, as expected
Slightly Larger Example

\[\{P : i + 1 \geq 0 \land i - j \geq 0\}\]
\[\downarrow (i + 1, i - j)\]

\(c_1 :\) \textbf{assume } j \geq i

\(c_2 :\) \(i := i - 1;\)
\[\downarrow (i + 1, i + 1)\]

Our verification condition is:

\[P \rightarrow \text{wlp}(c_1; c_2, (i + 1, i + 1) <_2 (i' + 1, i' - j'))[i/i', j/j']\]

Computing the \text{wlp}:

\[\text{wlp}(c_1; c_2, (i + 1, i + 1) <_2 (i' + 1, i' - j'))\]

\[\leftrightarrow \text{wlp}(c_1, \text{wlp}(i := i - 1, (i + 1, i + 1) <_2 (i' + 1, i' - j'))))\]

\[\leftrightarrow \text{wlp}(\textbf{assume } j \geq i, (i - 1 + 1, i - 1 + 1) <_2 (i' + 1, i' - j'))))\]

\[\leftrightarrow j \geq i \rightarrow (i, i) <_2 (i' + 1, i' - j')\]
\[
\{P : i + 1 \geq 0 \land i - j \geq 0\} \\
\downarrow (i + 1, i - j)
\]

c_1 : \ \textbf{assume} \ j \geq i

c_2 : \ i := i - 1;
\downarrow (i + 1, i + 1)

Our verification condition is now:

\[
P \rightarrow (j \geq i \rightarrow (i, i) <_2 (\iota' + 1, \iota' - \jota'))[\iota'/\iota', \jota'/\jota']
\]

\[
\leftrightarrow P \rightarrow (j \geq i \rightarrow (i, i) <_2 (i + 1, i - j))
\]

\[
\leftrightarrow i + 1 \geq 0 \land i - j \geq 0 \rightarrow (j \geq i \rightarrow (i, i) <_2 (i + 1, i - j))
\]

\[
\leftrightarrow i + 1 \geq 0 \land i - j \geq 0 \land j \geq i \rightarrow (i, i) <_2 (i + 1, i - j)
\]

\[
\leftrightarrow i > 0 \land i = j \rightarrow (i, i) <_2 (i + 1, i - j)
\]

Again, valid as expected
Example: Loops

```plaintext
{year = 1980}
while(days > 365)
  ↓ days
  {
    if(¬is_leap_year(year)) {
      days := days - 365;
      year := year + 1;
    } else {
      if(days > 366) {
        days := days - 366;
        year := year + 1;
      }
    }
  }
return year;
```
Example: Loops

First basic path:

\[
\begin{align*}
\{days > 0\} \\
\downarrow days \\
c_0 & : \textbf{assume }days > 365 \\
c_1 & : \textbf{assume }\neg \text{is\_leap\_year}(year) \\
c_2 & : days := days - 365 \\
c_3 & : year := year + 1 \\
\downarrow days
\end{align*}
\]

Our verification condition is:

\[
days > 0 \rightarrow \text{wp}(c_0; c_1; c_2; c_3, days < days')[days/days'] = days > 365 \land \neg \text{leap\_year}(year) \rightarrow days - 365 < days
\]

This is valid
Example: Loops

Second basic path:

\[
\begin{align*}
\{&days > 0\} \\
\downarrow &days \\
&c_0 : \textbf{assume} \ days > 365 \\
&c_1 : \textbf{assume} \ \text{is\_leap\_year}(\text{year}) \\
&c_2 : \textbf{assume} \ days > 366 \\
&c_3 : \text{days} := \text{days} - 366 \\
&c_4 : \text{year} := \text{year} + 1 \\
\downarrow &days
\end{align*}
\]

Our verification condition is:

\[
\begin{align*}
wlp(c_0; c_1; c_2; c_3; c_4, days < days')[days/days'] \\
= \quad \text{days} > 366 \land \text{leap\_year}(\text{year}) \rightarrow \text{days} - 366 < \text{days}
\end{align*}
\]

This is valid
Example: Loops

Third basic path:

\[
\begin{align*}
\{ \text{days} > 0 \} \\
\downarrow \text{days} \\
\text{c}_0 : \text{assume} \ \text{days} > 365 \\
\text{c}_1 : \text{assume} \ \text{is\_leap\_year(year)} \\
\text{c}_2 : \text{assume} \ \text{days} \leq 366 \\
\downarrow \text{days}
\end{align*}
\]

Our verification condition is:

\[
\text{wlp}(c_0;c_1;c_2, \text{days} < \text{days'})[\text{days'/days'}] \\
= \text{days} > 365 \land \text{days} \leq 366 \land \text{leap\_year(year)} \rightarrow \text{days} < \text{days} \\
= \text{days} = 366 \land \text{leap\_year(year)} \rightarrow \text{days} < \text{days}
\]

This is not valid!
Zune “Leap Year” Bug

This bug happened on Zune MP3 players

Our verification condition tells us when it will occur:

\[ \text{days} = 366 \land \text{leap\_year(year)} \]

For Zune owners, this was Dec. 31, 2008,

Termination bugs aren’t always complicated
Fixing the Zune bug

```c
{year = 1980}
while (days > 365)
  ↓ days
  {
    if (!is_leap_year(year)) {
      days := days - 365;
      year := year + 1;
    } else {
      if (days \geq 366) {
        days := days - 366;
        year := year + 1;
      }
    }
  }
return year;
```
Confirming the fix

Third basic path, revisited:

\[
\begin{align*}
\{ & \text{days} > 0 \} \\
\downarrow & \text{days} \\
\text{c}_0 : & \text{assume } \text{days} \geq 365 \\
\text{c}_1 : & \text{assume } \text{isLeapYear(year)} \\
\text{c}_2 : & \text{assume } \text{days} \leq 366 \\
\downarrow & \text{days}
\end{align*}
\]

Our verification condition is:

\[
\text{wlp}(c_0; c_1; c_2, \text{days} < \text{days}')[\text{days}/\text{days}']
\]

\[=
\text{days} > 365 \land \text{days} < 366 \land \text{isLeapYear(year)} \rightarrow \text{days} < \text{days}
\]

\[=
\text{false} \land \text{isLeapYear(year)} \rightarrow \text{days} < \text{days}
\]

This is now valid
Example: Recursive Function

```plaintext
proc BinarySearch(a : array, l, u, e)
  pre u − l + 1 ≥ 0
  post true
  ↓ u − l + 1
  {
    if(l > u) return 0;
    else {
      m := (l + u)/2;
      if(a[m] = e) return 1;
      else if(a[m] < e) return BinarySearch(a, m + 1, u, e);
      else return BinarySearch(a, l, m − 1, e);
    }
  }
```
Example: Recursive Function

```plaintext
proc BinarySearch(a : array, l, u, e)
pre u - l + 1 \geq 0
post true
↓ u - l + 1
```

- ↓ \(u - l + 1\) maps environment to \(\mathbb{N}\)
- With each call, the interval \([l, u]\) shortens
- But if \(l > 0\), then \(u - l\) isn’t \(\mathbb{N}\); thus, \(u - l + 1\)
- So, need to prove that \(u - l + 1\) isn’t negative; thus, precondition
- Also need to prove that \(u - l + 1\) decreases with each call
Example: Recursive Function

First two basic paths:

\[
\begin{align*}
\{u - l + 1 \geq 0\} \\
\downarrow u - l + 1 \\
\text{assume } l > u; \\
rv := 0; \\
\downarrow u - l + 1
\end{align*}
\]

\[
\begin{align*}
\{u - l + 1 \geq 0\} \\
\downarrow u - l + 1 \\
\text{assume } l \leq u; \\
m := (l + u) \text{ div } 2; \\
\text{assume } a[m] = e; \\
rv := 1; \\
\downarrow u - l + 1
\end{align*}
\]

We don’t need to worry about these for termination

They end the recursion
Example: Recursive Function

The next basic path

\[
\begin{align*}
\{ & u - l + 1 \geq 0 \} \\
\downarrow & u - l + 1 \\
\text{assume} & l \leq u; \\
& m := (l + u) \text{ div } 2; \\
& \text{assume} a[m] \neq e \\
& \text{assume} a[m] < e \\
\downarrow & u - (m + 1) + 1
\end{align*}
\]

What’s the verification condition?

\[
u - l + 1 \geq 0 \land l \leq u \land a[(l + u) \text{ div } 2] \neq e \land a[(l + u) \text{ div } 2] < e \rightarrow \\
u - (((l + u) \text{ div } 2) + 1) + 1 < u - l + 1
\]
Example: Recursive Function

The next basic path

\[
\begin{align*}
\{ & u - l + 1 \geq 0 \} \\
\downarrow & u - l + 1 \\
\text{assume} & l \leq u; \\
& m := (l + u) \text{ div } 2; \\
\text{assume} & a[m] \neq e \\
\text{assume} & a[m] \geq e \\
\downarrow & (m - 1) - l + 1
\end{align*}
\]

What’s the verification condition?

\[
\begin{align*}
& \quad u - l + 1 \geq 0 \land l \leq u \land a[(l + u) \text{ div } 2] \neq e \land a[(l + u) \text{ div } 2] \geq e \rightarrow \\
& \qquad (((l + u) \text{ div } 2) - 1) - l + 1 < u - l + 1
\end{align*}
\]
We’ve seen how to:

1. Decompose annotated programs into **basic paths**
2. Generate **verification conditions** for basic paths
3. Solve VCs in selected **first-order theories**

This gives us **automatic verification** if:

1. Background first-order theory is **decidable**
2. The program’s annotations are **inductive**

Let’s assume research will continue to improve decision procedures

Where do we get the annotations from?
Research is also progressing on automatic invariant inference

- This is **static analysis**, and in particular, **abstract interpretation**
- Represent program states using **abstract domain**
- Examples: intervals, affine spaces, octagons, …
- “Interpret” program over abstract state space
- Collect facts from domain to generate assertions

This can provide loop index bounds, heap properties, …

**But** in general, human insight is required

We’ll look at a strategy for getting to these insights
Basic Facts

To develop a proof of correctness, we need:

- Loop invariants that are strong enough for the postcondition
- Ranking functions that agree with our program

Start by looking for obvious **basic facts** to add to loop invariants

- Initialization conditions
- Loop index bounds
- Index updates
- Guard conditions
- Array bounds, non-constant divisors

Pay special attention to things related directly to the specification
Basic Facts: Linear Search

\[
i := l; \\
\textbf{while}(i \leq u) \{ \\
\quad \textbf{if}(a[i] = e) \text{ return } 1; \\
\quad i := i + 1; \\
\}
\]

- **Initialization:** \(i := 1\)
- **Loop guard:** \(i \leq u\)
- **Update:** \(i := i + 1\)

These facts give us a basic annotation:

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

Why \(u + 1\) in the upper bound?
\begin{equation*}
    i := |a| - 1;
    \textbf{while} (i > 0) \{ \\
        j := 0;
        \textbf{while} (j < i) \{ \\
            \textbf{if} (a[j] > a[j + 1]) \{ \\
                t := a[j];
                a[j] := a[j + 1];
                a[j + 1] := t;
            \}
        \}
    i := i - 1;
    \}
\end{equation*}

Bounds on the outer loop?
\[-1 \leq i < |a|\]

Inner loop for \(i\)?
\[0 < i < |a|\]

Inner loop for \(j\)?
\[0 \leq j \leq i\]
Usually, basic facts aren’t enough to prove a postcondition. Heuristically, we can compute preconditions to strengthen facts:

1. Find an annotation $F$ not supported by earlier annotations.
2. Compute weakest preconditions of $F$, ending at:
   - Loop invariants
   - Procedure entry point
3. At each new annotation, generalize resulting facts as necessary.

**Observe:** This is similar to the “working backwards” approach we used to prove properties using the operational semantics.
Precondition Method: Linear Search

\[ i := l; \]
\[ \textbf{while}(i \leq u) \]
\[ \{ l \leq i \leq u + 1 \} \]
\[ \{ \]
\[ \textbf{if}(a[i] = e) \textbf{return} 1; \]
\[ i := i + 1; \]
\[ \} \]
\[ \textbf{return} 0; \]

Recall that our postcondition was:

\[ (rv = 1) \leftrightarrow \exists i. l \leq i \leq u \land a[i] = e \]
Consider the basic path:

\[
\begin{align*}
\{l \leq i \leq u + 1\} \\
\text{assume } i > u; \\
rv := 0; \\
\{(rv = 1) \iff (\exists i. l \leq i \leq u \land a[i] = e)\}
\end{align*}
\]

The verification condition is (after simplification):

\[
l \leq i \leq u + 1 \land i > y \rightarrow (\forall j. l \leq j \leq u \leftarrow a[j] \neq e)
\]

The antecedent says nothing about \( a \): can’t prove \textbf{post}

This tells us the invariant needs to be strengthened
We want to strengthen:

\[ I : l \leq i \leq u + 1 \]

to prove:

\[
\{ I : l \leq i \leq u + 1 \} \\
\text{assume } i > u; \\
rv := 0; \\
\{ Q : (rv = 1) \leftrightarrow (\exists i. l \leq i \leq u \land a[i] = e) \}
\]

We propagate the postcondition backwards using \( \text{wlp} \):

\[
\text{wlp}(\text{assume } i > u; rv := 0, Q) = i > u \rightarrow (\forall j. l \leq j \leq u \rightarrow a[j] \neq e)
\]

This is what we need for the postcondition to hold in the \textit{false} case

Our invariant will need to generalize to the \textit{true} case as well
Precondition Method: Linear Search

Let’s try propagating this backwards through the loop once:

\[
\{ H \} \\
\textbf{assume} \ i \leq u; \\
\textbf{assume} \ a[i] \neq e; \\
i := i + 1; \\
\{ I : i > u \rightarrow (\forall j. l \leq j \leq u \rightarrow a[j] \neq e) \} \\
\]

\[
\text{wlp}(c_0; c_1; c_2, I) = i \leq u \rightarrow (a[i] \neq e \rightarrow I[i + 1/i]) \\
= i \leq u \land a[i] \neq e \land i + 1 > u \rightarrow (\forall j. l \leq j \leq u \rightarrow a[j] \neq e) \\
= i = u \land a[i] \neq e \rightarrow (\forall j. l \leq j \leq u \rightarrow a[j] \neq e) \\
= i = u \land a[i] \neq e \rightarrow (\forall j. l \leq j < u \rightarrow a[j] \neq e) \\
\]
Generalizing Facts

So far, we’ve obtained the following facts by working backwards:

\[
\begin{align*}
  i > u & \rightarrow (\forall j. l \leq j \leq u \rightarrow a[j] \neq e) \\
  i = u \land a[i] \neq e & \rightarrow (\forall j. l \leq j < u \rightarrow a[j] \neq e)
\end{align*}
\]

Using a bit of insight, we see the following is relevant to both facts:

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

This makes sense to add to our loop invariant

1. It captures the necessary functionality of the code
2. It’s related to the progress of the loop

We can generalize this insight:

*replace fixed terms with ones that evolve with the loop*
Example: Binary Search

```plaintext
proc BinarySearch(a : array, l, u, e)
  pre true
  post true
{ 
  if(l > u) return 0;
  else {
    m := (l + u)/2;
    {0 ≤ m < |a|}
    if(a[m] = e) return 1;
    {0 ≤ m < |a|}
    if(a[m] < e) return BinarySearch(a, m + 1, u, e);
    else return BinarySearch(a, l, m - 1, e);
  }
}
```

We want to derive a precondition to satisfy annotations.
Example: Binary Search

Consider the basic path ending at the first annotation

\[
\begin{align*}
\{P\} \\
\text{assume } l \leq u; \\
m := (l + u) \div 2; \\
\{F : 0 \leq m < |a|\}
\end{align*}
\]

Working backwards with \( \text{wlp} \),

\[
\begin{align*}
\text{wlp}(c_0; c_1, 0 \leq m < |a|) &= \text{wlp}(c_0, \text{wlp}(m := (l + u) \div 2, 0 \leq m < |a|)) \\
&= \text{wlp}(\text{assume } l \leq u, 0 \leq (l + u) \div 2 < |a|) \\
&= l \leq u \rightarrow 0 \leq (l + u) \div 2 < |a|
\end{align*}
\]

We see that:

\[
0 \leq l \land u < |a| \Rightarrow l \leq u \rightarrow 0 \leq (l + u) \div 2 < |a|
\]

\( 0 \leq l \land u < |a| \) will give us the runtime assertions
Use this approach when you’re working in Dafny

1. Write your postcondition first
2. Identify the basic facts needed to run the code
3. List the basic paths in your method
4. Compute preconditions to propagate facts to invariants
5. Sanity-check your VCs when the verifier complains

This will save you lots of time and stress!
Next Lecture

We’ll finish up strategies for developing inductive annotations

The next assignment will go out later today

Look at it early

Come to office hours with questions soon