Program Logic
Please Read

• Read chapter 4 of Huth & Ryan
• First part of Assignment 7 (to be posted):
  • Exercises 4.3, p 300-302:
    • 5.c
    • 14 Multi1
    • 17 Div
  • Exercises 4.4, p 303
    • 1.b
    • 1.f
Turing may have been the first to consider proving that a program is correct, in his 1949 paper (3 typewritten pages):

“Checking a Large Routine”

“How can one check a large routine in the sense that it's right?

… make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.”


http://www.turingarchive.org/viewer/?id=462&title=01
Robert W. Floyd

“Assigning meanings to programs”, 1967

Floyd attributed the idea to Saul Gorn in part.

FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ ($n \geq 0$)
One Type of Verification

• In the present slides, we focus on one type of verification, known as “inductive assertions”.

• Another type is based on structural induction and is relevant for functional programs with list data structures. We don’t discuss it here, but it is the basis for the famous ACL2 prover by Boyer and Moore.
Caution: Informality

• We will assume certain properties of integers, etc. are provable, but do not work within a specific formal theory for this aspect.

• The focus is more on program structure.
Program Graph Model

• The first model is conceptual.
• It will enable us to express some basic ideas, that will be used in more typical models.
• For now, we assume imperative programs, without recursion.
• [This is a variant on a model presented in R.M. Keller, Formal Verification of Parallel Programs, CACM 1976. Downloadable here: http://dl.acm.org/citation.cfm?id=360251]
Program Graph Model

• A program is a directed graph with labeled arcs.
• The nodes represent possible stopping points between changes.
• The arcs represent two things:
  • Gate predicate: indicating when the arc can be taken.
  • Assignment function: indicating change in values of zero or more program variables if the arc is taken.
Example

Gates are shown in [brackets]. If none is shown, the gate is implicitly T.

Assignments are shown with :=. If none is shown, the assignment function is implicitly the identity function on all variables. The RHS expressions are computed first, then assigned all at once to the LHS variables.
Example

Gates are shown in [brackets]. If none is shown, the gate is implicitly T.

Assignments are shown with :=. If none is shown, the assignment function is implicitly the identity function on all variables. The RHS expressions are computed first, then assigned all at once to the LHS variables.
Semantics

• A program represents a labeled-transition system.

• The **states** of a system consists of:
  • A single node name, representing the node before the next arc to be taken, if any.
  • A value for each variable.
Some States of the Previous Example

These states represent the sequence of steps taken if the program is started in node 1 with $n = 3$, assuming $i$ and $f$ are initially 0.

What states would occur if the program is started in node 1 with $n = 5$? with $n = 0$?
Semantics of the Arcs

• The arcs of a program express state-transition semantics.

• That is, each arc determines transitions as follows, where $v$ stands for the vector of all variables.

$$[P(v)] v := F(v)$$

$$(m, v) \rightarrow (n, F(v)), \text{ provided that } P(v) \text{ holds}$$
Composite Semantics

• Consider composing two consecutive arcs into one:

\[
\begin{align*}
\text{[}P_{ab}(v)\text{]} & \quad v := F_{ab}(v) \\
\text{[}P_{bc}(v)\text{]} & \quad v := F_{bc}(v)
\end{align*}
\]

\[
\begin{align*}
P_{ac}(v) &= P_{ab}(v) \land P_{bc}(F_{ab}(v)) \\
F_{ac}(v) &= F_{bc}(F_{ab}(v))
\end{align*}
\]
Example of Composite Semantics

• Here all gates are T, so are not shown

\[ a \xrightarrow{f := f \ast i} b \xrightarrow{i := i + 1} c \]

\[ a \xrightarrow{(i, f) := (i + 1, f \ast i)} \]
Example of Composite Semantics

• Note that reversing the assignments is not generally equivalent.

\[
\begin{align*}
  i &:= i+1 \\
  f &:= f*i \\
  b &\rightarrow c \\
  a &\rightarrow b \\
  a &\rightarrow c
\end{align*}
\]

\[
\begin{align*}
  (i, f) &:= (i+1, f*(i-1)) \\
  c &\rightarrow b \\
  a &\rightarrow c
\end{align*}
\]
Assertions

- Assertions are logical expressions that annotate nodes.

- They describe a condition that is supposed to be true when that node is the state component.
Example with Assertions

Assertion

1. \( A_1: n \geq 0 \)
2. \( (i, f) := (1, 1) \)
3. \([i \leq n] \ (i, f) := (i+1, f*i) \)
4. \( A_2: i \leq n+1 \land f = (i-1)! \land n \geq 0 \)
5. \([i > n] \)
6. \( A_3: f = n! \)
Executable Assertions

• You may be familiar with executable `assert` statements.

• These are used in development to indicate relations among variables hold, as if the program should not continue if they don’t.

• If the assertion does not hold, the program terminates immediately, indicating the source location.
Proving Assertions

• Verification of a program consists of proving that the assertions are valid, without executing the program.

• The assertion at the initial node of the program is assumed to be true, so we call it The Assumption.

• The assertion at the final node is desired to be true, so we call it The Expectation.
Verification Conditions

• Once assertions have been assigned, the proof can be accomplished by a set of detachable formulas, one for each arc.

• We call these Verification Conditions (VCs).

• If the assertions are correct, the VCs can be proved without reference to the original program.
Uniform Verification Conditions for the Graph Model

• Recall the semantics of arcs: There is a transition 
  \((m, v) \rightarrow (n, F(v))\), provided that \(P(v)\) holds

• The corresponding VC given assertions \(A_m\) and \(A_n\) is thus:

\[
VC_{mn}: A_m(v) \land P(v) \rightarrow A_n(F(v))
\]
Example VC’s

• $\text{VC}_{mn}: A_m(v) \land P(v) \rightarrow A_n(F(v))$

• $\text{VC}_{12}: A_1(i, f, n) \land T \rightarrow A_2(1, 1, n)$

• $\text{VC}_{22}: A_2(i, f, n) \land i <= n \rightarrow A_2(i+1, f^*i, n)$

• $\text{VC}_{23}: A_2(i, f, n) \land i > n \rightarrow A_3(i, f, n)$
What are the assertions?

• The assertion at the entry node of the program is given as a restriction on initial values. I call it 
  **The Assumption.**

• The assertion at the extra node of the program is given as the desired result. I call it 
  **The Expectation.**
Example Assumption and Expectation

• The example program was designed to compute the factorial of a natural number.

• Therefore:
  Assumption: $n > 0$

  Expectation: $f = n!$  (i.e. $1*2*3*...*n$)
  note: $0! = 1$
The Boundary Assertions are Given

• Therefore we know:

\[ A_1(i, f, n): n \geq 0 \quad \text{Assumption} \]

\[ A_3(i, f, n): f = n! \quad \text{Expectation} \]
The remaining assertions must be created/discovered.

\[ A_2(i, f, n) \]

This assertion must express:

**what is known** about the state of the program at the indicated node based on the Assumption at entry, and **what must be true** to arrive at the exit with the Expectation met.
What we have so far

Assertions

1. \( A_1(i, n, f): n \geq 0 \)
   
   \((i, f) := (1, 1)\)

2. \( A_2(i, n, f) \)
   
   \([i \leq n] \ (i, f) := (i+1, f*i)\)

3. \( A_3(i, n, f): f = n! \)

\( A_2(i, n, f) \) should include:

\[ f = (i-1)! \] because we observe \( i = n+1 \) upon exit.
Is $A_2(i, n, f)$: $f = (i-1)!$ enough?

- $VC_{12}$: $n \geq 0 \land T \rightarrow A_2(1, 1, n)$

- $VC_{22}$: $A_2(i, f, n) \land i \leq n \rightarrow A_2(i+1, f*i, n)$

- $VC_{23}$: $A_2(i, f, n) \land i > n \rightarrow f = n!$
Is $A_2(i, n, f): f = (i-1)!$ enough?

• Below are the VC’s with the assertions substituted.

  • $VC_{12}: n \geq 0 \land T \rightarrow 1 = (1-1)!$

  • $VC_{22}: f = (i-1)! \land i \leq n \rightarrow f \times i = ((i+1)-1)!$

  • $VC_{23}: f = (i-1)! \land i > n \rightarrow f = n!$

• Are these provable?
Are the VC’s provable?

• $\text{VC}_{12}$: $n \geq 0 \land T \rightarrow 1 = (1-1)!$

• reduces to $n \geq 0 \rightarrow 1 = 0!$

• Provable (assuming appropriate axioms for number), since $1 = 0!$
Are the VC’s provable?

- $\text{VC}_{22} \colon f = (i-1)! \land i \leq n \rightarrow f \ast i = ((i+1)-1)!$

- reduces to
  
  $f = (i-1)! \land i \leq n \rightarrow f \ast i = i!$

- If $f = (i-1)!$ then $f \ast i = i \ast (i-1)!$, and indeed $i \ast (i-1)! = i!$

- So yes, provable.
Are the VC’s provable?

• $\text{VC}_{23}: f = (i-1)! \land i > n \rightarrow f = n!$

• This VC is \textit{not} provable.

• We only have $i > n$, not $i = n+1$.

• How can we get $i = n+1$ on exit?
Revisit the assertions

A₁(i, n, f): n ≥ 0

(i, f) := (1, 1)

[i <= n] (i, f) := (i+1, f*i)

A₂(i, n, f): f = (i-1)!

[i > n]

A₃(i, n, f): f = n!

A₂(i, n, f) should include:

f = (i-1)! and i ≤ n+1.
AsserLons

Revised

\[ (i, f) := (1, 1) \]

\[ [i \leq n] \quad (i, f) := (i+1, f*i) \]

\[ [i > n] \]

1. \( A_1(i, n, f): n \geq 0 \)
   \( (i, f) := (1, 1) \)

2. \[ i \leq n \] \( (i, f) := (i+1, f*i) \)
   \( A_2(i, n, f): f = (i-1)! \land i \leq n+1 \)
   \[ [i > n] \]

3. \( A_3(i, n, f): f = n! \)
Is $A_2(i, n, f): f = (i-1)! \land i \leq n+1$ enough?

- Below are the VC’s with the assertions substituted.
- $VC_{12}: n \geq 0 \land T \rightarrow 1 = (1-1)! \land 1 \leq n+1$
- $VC_{22}: f = (i-1)! \land i \leq n+1 \land i \leq n \rightarrow f* = ((i+1)-1)! \land i+1 \leq n+1$
- $VC_{23}: f = (i-1)! \land i \leq n+1 \land i > n \rightarrow f = n!$
- Are these provable?
Are the revised VC’s provable?

- \( VC_{12} : n \geq 0 \land T \rightarrow 1 = (1-1)! \land 1 \leq n+1 \)
- reduces (again assuming some properties of numbers) to

\[
n \geq 0 \rightarrow 1 = 0! \land 1 \leq n+1
\]

- This is provable, since \( 1 = 0! \), and \( n \geq 0 \rightarrow 1 \leq n+1 \).
Are the revised VC’s provable?

• **VC$_{22}$**: $f = (i-1)! \land i \leq n+1 \land i \leq n$

  $\implies f*i = ((i+1)-1)! \land i+1 \leq n+1$

• As $i \leq n$ implies $i \leq n+1$, the latter is subsumed (redundant).

• With some simplification, this VC reduces to:

  $f = (i-1)! \land i \leq n \implies f*i = i! \land i+1 \leq n+1$

• which is easily proved.
Are the revised VC’s’s provable?

• $\text{VC}_{23}: f = (i-1)! \land i \leq n+1 \land i > n \rightarrow f = n!$

• Because we are assuming natural numbers, we can show: $i \leq n+1 \land i > n \iff i = n+1$.

• Therefore, this VC reduces to: $f = (i-1)! \land i = n+1 \rightarrow f = n!$ which can be proved.
Soundness and Completeness

We can use some of the same terminology from logic to express phenomena of assertions:

- The set of assertions is **sound** if they are in fact valid for any execution of the program.

- The set of assertions is **complete** if they are sufficient to **prove** the verification conditions.
Conclusion: These assertions are sound and complete.

Assertions

1. $A_1(i, n, f): n \geq 0$
   
   $(i, f) := (1, 1)$

2. $A_2(i, n, f): f = (i-1)! \land i \leq n+1$
   
   $[i \leq n] (i, f) := (i+1, f \cdot i)$

3. $A_3(i, n, f): f = n!$
   
   $[i > n]$
Checking Assertions by Execution

• We can do a *sanity check* on the soundness of our assertions by running the program for various input data.

• This is only a sanity check. It proves neither soundness nor completeness. It can at best increase our confidence.
Sanity Check: Python Program Execution

def fac(n):
    (i, f) = (1, 1)
    while i <= n:
        (i, f) = (i+1, f*i)
    return f

def fac_with_assertions(n):
    assert n >= 0        #A1, assumption
    (i, f) = (1, 1)
    assert i <= n+1 and f == pyfac(i-1)    # A2
    while i <= n:
        assert i <= n+1 and f == pyfac(i-1)    # A2
        (i, f) = (i+1, f*i)
    assert f == pyfac(n)    #A3, expectation
    return f

# pyfac is a different way of computing fac
def pyfac(n):
    return reduce(multiply, range(1, n+1), 1)

def multiply(x, y):
    return x*y

Comparing n, fac(n), and pyfac(n):

<table>
<thead>
<tr>
<th>n</th>
<th>fac(n)</th>
<th>pyfac(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>720</td>
<td>720</td>
</tr>
<tr>
<td>7</td>
<td>5040</td>
<td>5040</td>
</tr>
<tr>
<td>8</td>
<td>40320</td>
<td>40320</td>
</tr>
<tr>
<td>9</td>
<td>362880</td>
<td>362880</td>
</tr>
<tr>
<td>10</td>
<td>3628800</td>
<td>3628800</td>
</tr>
</tbody>
</table>
Loop Invariants

• A2 has a special role and name: It is called the loop invariant, because:
  • It is true **before every test** of the loop condition, including:
    • the first time, and
    • the last time.
  • It is true **at the end** of the loop body.
• As we shall see, discovering loop invariants is the most creative part of program verification.
Partial Automation of Assertion Derivation

• Given one assertion on a node, the assertion on the other connected but otherwise isolated node can be derived.
  \[ VC_{mn} : A_m(v) \land P(v) \rightarrow A_n(F(v)) \]

• If \( A_m \) is known, then the strongest possible \( A_n \) is seen to be \( A_n(v') : \exists v [A_m(v) \land P(v) \land v'=F(v)] \). This is called the strongest post-condition corresponding to \( A_m \).

• However it is not so useful, due to the \( \exists \), which “loses information”.

Weakest Pre-Condition

- Given one assertion on a node, the assertion on the other connected but otherwise isolated node can be derived.

\[ V_{C_{mn}}: A_m(v) \land P(v) \rightarrow A_n(F(v)) \]

- If \( A_n \) is given, then the \textit{weakest possible} \( A_m \) is seen to be

\[ A_m(v): P(v) \rightarrow A_n(F(v)). \]

- This is called the \textit{weakest pre-condition} corresponding to \( A_n \).

- Denote it as: \( WP[A_n(v), v := F(v), P(v)] \)
Weak vs. Strong Assertions

• If \( A \rightarrow B \) but not \( (A \rightarrow B) \) then
  • A is stronger than B.
  • B is weaker than A.
  • A conveys more information than B.
• The **weakest precondition** is the precondition that implies any other valid precondition.
• The **strongest assertion ever** is \( \bot \) ("bottom", false). It conveys so much information that *everything* is derivable from it.
• The **weakest assertion ever** is \( T \) ("top", true). It conveys no information. Nothing other than \( T \) is derivable from it.
Weakest Pre-Condition (WP)

• All you need to remember is in this box:

\[
\text{WP}[\text{A}_n(v), v := F(v), P(v)]
\]

\[
\text{P}(v) \rightarrow \text{A}_n(F(v))
\]
Special Case when $P = T$

• All you need to remember is in this box:

\[ WP[A_n(v), v := F(v), P(v)] \]

\[ [T] v := F(v) \]

\[ A_n \]

[Diagram]

\[ WP[A_n(v), v := F(v), T] \]

*is just*

\[ A_n(F(v)) \]
Example of WP

• Consider the fragment shown below, where $A_c$ is given as shown.

• We derive $A_b$ as $WP[A_c, i := i+1, T]$

\[ \overset{b}{i := i+1} \quad A_b \text{ is } WP[A_c, i := i+1, T]
\]

\[ \overset{c}{\text{A}_c(i, f, n) : f = (i-1)! \land i \leq n+1} \]

\[ \overset{T}{\text{which is: } T \rightarrow A_c(i+1, f, n)} \]

\[ \overset{T}{\text{which is } T \rightarrow f = ((i+1)-1)! \land i+1 \leq n+1} \]

\[ \overset{T}{\text{which simplifies to } f = i! \land i \leq n} \]
Another Example of WP

• Consider the fragment shown below, where $A_b$ is given as shown.
• We derive $A_a$ as $WP[A_b, f := f*i, i <= n]$

\[ A_a \text{ is } WP[A_b, f := f*i, i <= n] \]

\[ \text{which is: } i <= n \rightarrow A_b(i, f*i, n) \]

\[ \text{which is } i <= n \rightarrow f = i! \land i \leq n \]

\[ A_b(i, f, n): f = i! \land i \leq n \]
Chaining Weakest Pre-Conditions

- The weakest pre-condition allows us to work backward through a chain of arcs.

- This is similar to the composite semantics idea discussed earlier, except now we are deriving assertions, not semantics.
Working backward with WP

\[ a \]

\[ f := f \times i \]

\[ b \]

\[ i := i + 1 \]

\[ c \]

\[ f = (i-1)! \land i \leq n+1 \]
Working backward with WP

a

f := f*i

b

f = i! ∧ i+1 ≤ n+1

i := i+1

c

f = (i-1)! ∧ i ≤ n+1
Working backward with WP

\[ f \cdot i = i! \land i+1 \leq n+1 \]

\[ f := f \cdot i \]

\[ f = i! \land i+1 \leq n+1 \]

\[ i := i+1 \]

\[ f = (i-1)! \land i \leq n+1 \]
Derive the missing assertions

\[ y := x \]

\[ x := yz - 1 \]

\[ x > y \]
Sequencing Formula

• Suppose that the semicolon in Fragment1 ; Fragment2 means that the two fragments are executed in sequence.

• The backward WP reasoning used in previous slides can be summarized as:

  \[ \text{WP}[A, \text{Fragment1} ; \text{Fragment2}, T] = \text{WP}[\text{WP}[A, \text{Fragment2}, T], \text{Fragment1}, T] \]

  where A is the post-condition after Fragment2.

• The next slide illustrates this important formula.
WP for Sequenced Fragments

WP[A, Fragment1; Fragment2, T] =

WP[A, Fragment1; Fragment2, T] =

WP[WP[A, Fragment2, T], Fragment1, T]

WP[A, Fragment2, T]

WP[A, Fragment2, T]

A
Two Options for Sequenced Fragments

• We now have two options for deriving WP for a sequence of fragments.
  • Derive the composition of the fragments, then derive the WP in one step.
  • Derive the WP stepwise, by using the WP of a later fragment in the WP of its predecessor.
Branching

• When the P part of an arc is not identically T, there is usually a paired arc with the complementary condition.

• This occurs in conditionals and in iteration.

• In this case, both WP’s need to be satisfied, so the WP is the conjunction of the two.
Branching

In this case, both WP’s need to be satisfied, so the WP at 1 is the **conjunction** of the two.

\[
\text{WP: } P(v) \rightarrow A_m(F(v)) \land \neg P(v) \rightarrow A_n(G(v))
\]

- \([P] v := F(v)\)
- \([\neg P] v := G(v)\)
Branching

• We could also express WP for 2-way branching using a 3-ary connective __ ? __ : __ as in C, C++, Java:

In Python, WP would be the expression

\[ A_m(F(v)) \text{ if } P(v) \text{ else } A_n(G(v)) \]
Looping

• Consider the form of a while loop
  while P do v := F(v)
Looping

- We temporarily “unwind” this loop.
- We then have as $A_m$
  \[
  \text{WP}[A_n, \text{noop}, \neg P] \land \text{WP}[A_m, v := F(v), P]
  \]
  but this is recursive.
Loop Invariant

• Because $A_m = WP[A_n, \text{noop}, \neg P] \land WP[A_m, v := F(v), P]$ is recursive, and does not necessarily have a closed form, we try to discover a loop invariant $I$ that satisfies the equation for $A_m$.

• $I = WP[A_n, \text{noop}, \neg P] \land WP[I, v := F(v), P]$

• Sufficient conditions for this equation to be satisfied are:

  $$I \rightarrow WP[I, v := F(v), P]$$

  $$I \rightarrow WP[A_n, \text{noop}, \neg P]$$
Loop Invariant

• We can rewrite the sufficient conditions for the loop invariant $I$:
  
  $I \land P \rightarrow WP[I, \nu := F(\nu), T]$
  
  $I \land \neg P \rightarrow A_n$
Refolding the loop and considering initialization

• Loop Verification Conditions \( I \) are

\[
\begin{align*}
A_r &\rightarrow WP[I, v := G(v), T] \quad \text{Initialization} \\
I \land P &\rightarrow WP[I, v := F(v), T] \quad \text{Looping} \\
I \land \neg P &\rightarrow A_n \quad \text{Finalization}
\end{align*}
\]
Example

• Consider the “squaring by addition” program.
• Note that
  • $1^2 = 1$
  • $2^2 = 1 + 3$
  • $3^2 = 1 + 3 + 5$
  • $4^2 = 1 + 3 + 5 + 7$
    etc.
So we can square by adding odd numbers.
Squaring Program

1. \( A_1(i, j, s, n): n \geq 0 \)
   \( (i, j, s) := (1, 1, 0) \)

2. \( A_2(i, j, s, n): \text{TBD} \)
   \( [i \leq n] (i, j, s) := (i+1, j+2, s+j) \)

3. \( A_3(i, j, s, n): s = n^2 \)
   \( [i > n] \)

[i <= n] (i, j, s) := (i+1, j+2, s+j)
VC’s for the Squaring Program

• Loop Verification Conditions I are
  \[ n \geq 0 \rightarrow WP[I, (i, j, s) := (1, 1, 0), T] \]  \text{Init}
  \[ I \land i \leq n \rightarrow WP[I, (i, j, s) := (i+1, j+2, s+j), T] \]  \text{Loop}
  \[ I \land i > n \rightarrow s = n^2 \]  \text{Final}

• What should I be?

• From the factorial example, it should include
  \[ i \leq n+1. \]

• Also include \( s = (i-1)^2. \)

• And we need something about \( j. \) (What?)
Invariant for Squaring Program

- Proposed I: $i \leq n+1 \land s = (i-1)^2 \land \ldots$
- Check **Final VC:**
  $I \land i > n \rightarrow s = n^2$
  $i \leq n+1 \land s = (i-1)^2 \land i > n \rightarrow s = n^2$ ??

Because values are integer,
  $i \leq n+1 \land i > n$ is equivalent to $i = n+1$, as in factorial.
So the above simplifies to
  $s = (i-1)^2 \land i = n+1 \rightarrow s = n^2$
which is obviously valid.

So we only have 2 more VC’s to check, as long as we don’t take away anything from the proposed invariant I.
What to say about j?

• It is maintaining the “next odd number”.
• But the invariant must be true when \((i, j, s) = (1, 1, 0)\).
• So perhaps \(j = 2*i - 1\), making the invariant:
  \[i \leq n+1 \land s = (i-1)^2 \land j = 2*i - 1\]
• Check the initial VC:
  \[
  n \geq 0 \rightarrow WP[l, (i, j, s) := (1, 1, 0), T]
  n \geq 0 \rightarrow 1 \leq n+1 \land 0 = (1-1)^2 \land 1 = 2*1 - 1
  \]
  which is valid.
Checking the Loop VC

• \( I \land i <= n \rightarrow WP[I, (i, j, s) := (i+1, j+2, s+j), T]\)

  with \( I \) as \( i \leq n+1 \land s = (i-1)^2 \land j = 2*i - 1 \)

• We need to verify:

\[
(i \leq n+1 \land s = (i-1)^2 \land j = 2*i - 1) \land i \leq n \rightarrow \\
i+1 \leq n+1 \land s+j = ((i+1)-1)^2 \land j+2 = 2*(i+1) - 1
\]

(the WP part)
Checking the Loop VC

• We need to verify:
  \[(i \leq n+1 \land s = (i-1)^2 \land j = 2i-1) \land i \leq n\]
  \[\rightarrow i+1 \leq n+1 \land s+j = ((i+1)-1)^2 \land j+2 = 2(i+1) - 1\]

• The LHS simplifies to:
  \[i \leq n \land s = (i-1)^2 \land j = 2i - 1\]

• The RHS simplifies to:
  \[i \leq n \land s+j = i^2 \land j = 2i - 1\]

• which, based on the LHS, further simplifies to
  
  \[(i-1)^2 + (2i - 1) = i^2\]
  which simplifies to \(T\).