Inferring invariants

- There is no fully general automation for inferring invariants (as there is for the weakest precondition of assignment statements).
- This is one of the things that makes totally automated verification difficult.
- Finding the right invariant is still a human intellectual activity.

Approaching Invariants through the “Back Door”

- Suppose $A$ is an assertion that we desire to be true after a while loop.
- Suppose the invariant $I$ is unknown.
- We do know that whatever $I$ is, it satisfies:

$$ (I \land \neg P) \rightarrow A $$

by comparing the template:

\[
\begin{align*}
{I} \text{ while } P \text{ do } & \{ I \land \neg P \} \\
{I} \text{ while } P \text{ do } & \{ A \}
\end{align*}
\]

Approximating $I$

- For starters, then, we could "over-approximate" $I$ by $A$.
- Then working backward through the body $B$, we derive the "weakest pre-condition" corresponding to $A$, denoted: $wp_B(A)$
- We also know that $(I \land \neg P) \rightarrow wp_B(A)$ so we can use $wp_B(A)$ as another approximation to $I$.
- Continuing, $I$ is approximated by $A \lor wp_B(A) \lor wp_B(wp_B(A)) \lor wp_B(wp_B(wp_B(A))) \lor ...$

Example

\[
\begin{align*}
i & := 0; \\
\text{while } i < n & \text{ do } \\
& i := i+1 \\
& \text{ od} \\
& \{ i = n \}
\end{align*}
\]

\[
wp_i := i+1 (i = n) \text{ is } i+1 = n \text{ which is the same as } i = n-1. \\
wp_i := (wp_B := i = n) \text{ is } i+1 = n-1 \text{ which is } i = n-2.
\]

So the continued approximation is:

\[
i = n \lor (i = n-1) \lor (i = n-2) \lor ...
\]

suggesting as the invariant $i \leq n$, which we know works.

Moreover, for $n \geq 0$, $i \leq n$ is implied by $i = 0$, which is also required.

Proof Using JAPE

WHERE DISTINCT $i$, $n$ IS

\[
\begin{align*}
i & := 0; \\
\{ i \leq n \}
\end{align*}
\]

\[
\text{while } i < n \text{ do } i := i+1 \text{ od} \\
\{ i = n \}
\]
Proof Steps

1. \( |x| = 0 \) \( |y| = 0 \) while \( i \leq i + 1 \) \( |x| = n \)

Tuple rule

...\n
2. \( i \leq i + 1 \)
3. \( i = i + 1 \) while \( i \leq i + 1 \) \( |x| = n \)

Proof of Initialization

1. \( |x| = 0 \) \( |y| = 0 \) \( |x| = n \)

...\n
2. \( i \leq i + 1 \)
3. \( i = i + 1 \) while \( i \leq i + 1 \) \( |x| = n \)

Expand the while

... body

Body partial correctness

1. \( |x| = 0 \) \( |y| = 0 \) while \( i \leq i + 1 \) \( |x| = n \)

2. \( i \leq i + 1 \)
3. \( i = i + 1 \) while \( i \leq i + 1 \) \( |x| = n \)

Proof of Initialization

1. \( |x| = 0 \) \( |y| = 0 \) \( |x| = n \)

...\n
2. \( i \leq i + 1 \)
3. \( i = i + 1 \) while \( i \leq i + 1 \) \( |x| = n \)

Body Partial Correctness

... body

Body partial correctness

1. \( |x| = 0 \) \( |y| = 0 \) while \( i \leq i + 1 \) \( |x| = n \)

2. \( i \leq i + 1 \)
3. \( i = i + 1 \) while \( i \leq i + 1 \) \( |x| = n \)

Body decreases variant

1. \( |x| = 0 \) \( |y| = 0 \) while \( i \leq i + 1 \) \( |x| = n \)

Set the Variant: \( n-i \)

... body

Body decreases variant

1. \( |x| = 0 \) \( |y| = 0 \) while \( i \leq i + 1 \) \( |x| = n \)

2. \( i \leq i + 1 \)
3. \( i = i + 1 \) while \( i \leq i + 1 \) \( |x| = n \)

Prove the Variant Continuation Condition

1. \( |x| = 0 \) \( |y| = 0 \) while \( i \leq i + 1 \) \( |x| = n \)

2. \( i \leq i + 1 \)
3. \( i = i + 1 \) while \( i \leq i + 1 \) \( |x| = n \)

Body decreases variant

1. \( |x| = 0 \) \( |y| = 0 \) while \( i \leq i + 1 \) \( |x| = n \)

2. \( i \leq i + 1 \)
3. \( i = i + 1 \) while \( i \leq i + 1 \) \( |x| = n \)

Prove Body Decreases Variant

1. \( |x| = 0 \) \( |y| = 0 \) while \( i \leq i + 1 \) \( |x| = n \)

2. \( i \leq i + 1 \)
3. \( i = i + 1 \) while \( i \leq i + 1 \) \( |x| = n \)
Loop exit implies final expectation

\[
\begin{align*}
20: & 0 \leq m - 6n \\
21: & m \\
22: & 1 \leq n \\
23: & n \\
24: & 0 \leq n - (m - 6n) = n \\
\end{align*}
\]

Can you identify all the parts?

Subtleties About Loop Invariants

- Can the following Ntuple be proved?

\[
\begin{align*}
(j := 1; & i := 0; y := 0) \\
(y &:= x \land n = 0) \\
\text{while } i < n \text{ do } \\
& y := y + j; \\
& j := j + 2; \\
& i := i + 1 \\
\od \\
(y &:= n^n)
\end{align*}
\]

Is the invariant truly invariant?

This cannot be proved. j is not even mentioned in the hypothesis.

What went wrong?
Important: Strength of Invariants

- Our invariant was “too weak” to enable its own inductive proof.
- In making an invariant stronger, we get more information from which to derive the post condition.
- At the same time, however, we have more to prove with a stronger invariant, so we don’t want to go too far.

The Lattice of Invariants

\begin{itemize}
  \item \( \checkmark \) very weak, implies nothing but itself, but implied by everything
  \item \( \uparrow \) other invariants, implied by \( I^* \), but too weak in themselves to enable inductive proof
  \item \( I^* \) an inductive invariant, “just right”, but not necessarily unique
  \item \( \downarrow \) implies
  \item \( \downarrow \) very strong, implies everything, but implied by nothing but itself
\end{itemize}

An Inductive Invariant

\[
(b \land n)
\]

(j := 1; i := 0; y := 0)

\[
y = i 	imes j = 2 	imes 0 \land b \land a \land n
\]

while \( i < n \)
do
  \[
y := y + j;
  j := j + 2;
  i := i + 1
\]
od

(y = n \times n)

Now the Body Proof Works

\[
\begin{align*}
\text{Init} & \quad b \land n \quad \text{assumption} \\
\text{Loop} & \quad \text{while } i < n \text{ do } \quad \text{assumption} \\
& \quad j := 1 \quad \text{assumption} \\
& \quad i := 0 \quad \text{assumption} \\
& \quad y := 0 \quad \text{assumption} \\
& \quad \text{od} \quad \text{assumption} \\
\text{Exit} & \quad y = n \times n \quad \text{assumption}
\end{align*}
\]

The Completed Proof (lines 1-24)

(From a different run, so there may be minor differences.)

\[
\begin{array}{c|c}
\text{Line} & \text{Value} \\
\hline
1 & j := 1 \\
2 & i := 0 \\
3 & y := 0 \\
4 & \text{while } i < 5 \text{ do } \\
5 & \quad j := j + 2 \\
6 & \quad i := i + 1 \\
7 & \text{od} \\
8 & y = 15 \\
9 & \text{end}
\end{array}
\]

The Completed Proof (lines 25-51)

\[
\begin{array}{c|c}
\text{Line} & \text{Value} \\
\hline
16 & y := 0 \\
17 & j := 2 \\
18 & i := 0 \\
19 & y := 0 \\
20 & \text{while } i < 5 \text{ do } \\
21 & \quad j := j + 2 \\
22 & \quad i := i + 1 \\
23 & \text{od} \\
24 & y = 15 \\
25 & \text{end}
\end{array}
\]
Verifying Array Programs

• Arrays present extra challenges and interesting issues.

• A useful dichotomy:
  - Programs with read-only arrays
  - Programs with modifiable arrays

Array Mathematics

• An array can be treated as if a function:
  - It maps indices into values.
  - E.g., a 1-dimensional array with dimension 10 maps \( \{0, ..., 9\} \) into values of the type stored in the array.
  - \( a[i] \) is the value of this function with argument \( i \)

Read-Only Array Example: Finding a zero element

\[ \{ n \geq 0 \land \text{length}(a) = n \} \]
\[ i := 0; j := n \]
\[ \{ n \geq 0 \land i \leq n \land j < n \rightarrow a[j] = 0 \} \]
while \( i < n \) do
  if \( a[i] = 0 \) then \( j := i \) else skip fi;
  \( i := i + 1 \) od
\[ \{ j < n \rightarrow a[j] = 0 \} \]

JAPE Type-in (for reference)

WHERE DISTINCT i, j, n, a IS
\[ \{ 0 \leq n \leq \text{length}(a) \} \]
\[ i := 0; j := n \]
\[ \{ 0 \leq i \leq n \land j < n \land a[j] = 0 \} \]
while \( i < n \) do
  if \( a[i] = 0 \) then \( j := i \) else skipfi;
  \( i := i + 1 \) od
\[ \{ j < n \rightarrow a[j] = 0 \} \]

Salient Parts of Proof
Initialization

Loop Body Partial Correctness
Variant Continuation and Decrease

Remainder

Without collapsing, the proof is about 74 lines.

Quantifiers

- Quantifiers are handy representing information about arrays, e.g.
  - $\forall i ((0 < i) \land (i < n)) \rightarrow a[i-1] \leq a[i]
  - $\exists i ((0 \leq i) \land (i < n) \land a[i] = 0)$

Read-Only Example with Quantifiers

A program whose correct working depends on values stored in the array.

\[
\begin{align*}
1 & : (0 \leq x \land x < \text{length}(a)) \land a(x) = 0 \\
2 & : 0 \leq i \land \text{length}(a) \land a[i] = 0
\end{align*}
\]

- The assumption is that the array contains a zero element.
- The expectation is to find the index of such an element.
- Termination critically depends on this assumption.
- The invariant says that the element such that $a[x] = 0$ is still to be found.

Start of Proof

Consequence for Initialization

Done with Initialization
On to the while loop

A subtlety arises in proving this invariant.
If the loop does not stop, it means $a[i] \neq 0$.
This is used to re-establish the invariant.

Body Partial Correctness

Use the assignment rule.
This generates a consequent implication.

It is tempting to try backward $\land$ Introduction now. Don’t!

Use Forward $\land$ Elimination

Now use $\exists$ Elimination, introducing $i_2$

Note that $i_2$ cannot equal $i$. Why?
We need to show this, to get the post-condition.

To get $i_2 \neq i$
- Set up a bifurcation using $i \leq i_2$, i.e. $i < i_2 \lor i = i_2$.
- Then use $\lor$ Elimination:

Now try backward $\land$ Introductions.
- What to use for $x$ in $\exists x$...?
i2 is the obvious choice for x

Bottom Half: We want a contradiction

How to pick _B1?

For _B1 use a[i] = 0

Status

Closure of the top half

Now use equality, to get 35 from 26 and 34.
Closure

What about termination?

• What to use for _M?

Completed proof, using several lemmas 1/3

Modifiable Arrays

• Arrays are like functions.

• Assigning to an array element is like creating a new function.

• The new function differs from the old in that one element may be different from before.
JAPE Array Modification Notation

- Jape Notation: \( a_{\oplus i} \rightarrow v \) is the array that is like \( a \) except that the value of \( a[i] \) is \( v \).
- \( (a_{\oplus i} \rightarrow v)[i] = v \) same index, new value
- \( (a_{\oplus i} \rightarrow v)[j] = a[j] \) if \( j \neq i \) different index, old value

Array Element Assignment

- Consider a triple
  \[
  \{ ?? \} \\
  a[i] := E \\
  \{ B \}
  \]
- If this is viewed mathematically as an array replacement:
  \[
  \{ ?? \} \\
  a := (a_{\oplus i} \rightarrow E) \\
  \{ B \}
  \]
- then it is obvious what ?? should be.

Isn’t it?

- If this is viewed mathematically as an array replacement:
  \[
  \{ ?? \} \\
  a := (a_{\oplus i} \rightarrow E) \\
  \{ B \}
  \]
- then it is obvious what ?? should be:
  \[
  B[(a_{\oplus i} \rightarrow E)/a] \\
  \]
- i.e. replace all free instances of \( a \) in \( B \) with \( (a_{\oplus i} \rightarrow E)/a \).

Example

- \[
  \{ ?? \} \\
  a[1] := 99 \\
  a[0] = 98 \land a[1] > 50
  \]
- \[
  \{ (a_{\oplus 1} \rightarrow 99)[0] = 98 \land (a_{\oplus 1} \rightarrow 99)[1] > 50 \} \\
  a := (a_{\oplus 1} \rightarrow 99) \\
  \{ a[0] = 98 \land a[1] > 50 \}
  \]

JAPE Example

- \[
  \{ ?? \} \\
  a[1] := 99 \\
  a[0] = 98 \land a[1] > 50
  \]
- \[
  \{ (a_{\oplus 1} \rightarrow 99)[0] = 98 \land (a_{\oplus 1} \rightarrow 99)[1] > 50 \} \\
  a := (a_{\oplus 1} \rightarrow 99) \\
  \{ a[1] = 99 \land a[0] = 98 \land a[1] > 50 \}
  \]

Bornat’s Array Assignment Rule in JAPE

What \( a[E] \) computes means:

- Formula B with each free \( a \) replaced with \( a_{\oplus E} \rightarrow F \)

- \[
  \{ a[E] \text{ computes} \} \\
  0 \leq E < \text{length}(a)
  \]

- \[
  a[E] := B \\
  \]

- \[
  \]
Simplifying Array Indexing Expressions

- JAPE Provides two equality rules for simplifying indexing expressions.
- These are needed for proving just about any array program. They are in the Indexing menu:

The first rule deals with the case where the index \( G \) on the "new" array is the same as the index \( E \) that was modified.

The second deals with the case where they are not the same.

One of these cases needs to be established to perform any simplification.

Simplification Example

- Prove
  \[
  \{a[0] = 98 \land a[1] = 5\} \implies \{a[0] = 98 \land a[1] > 50\}
  \]

Candidates for Simplification

Using Indexing Rule where \( E \neq G \)

Note that the entire indexed expression must be selected, not just the array portion.

Result

when \( 0 \neq 1 \), \( \{a[0] = 99\}[0] \) is same as \( a[0] \) whatever that happens to be.

Using Indexing Rule where \( E = G \)
Result

1. \( 1 = 0 \)
2. \( a[i] = \text{length}(a) \)
3. \( i = \text{length}(a) \)

where \( 1 = 1, (a[i = 99]) \) is same as RHS of assignment

Using Bounds Inference

Once an index is used, it is assumed to be valid.

Select only indexed expression, not entire statement.

Goal selected, and expression selected

Completion of Proof, using Equality

(complex)

Rule

A = A

Goal selected, and expression selected

Completed Proof
Consecutive Array Modifications

- Could simplify array-modification expressions as soon as they arise, or
- Wait, and deal with nested expressions.
- The first is probably better.

Consecutive Modification Example

\[
\begin{align*}
\text{\texttt{a[i] = 0}} \\
\text{\texttt{a[i] := a[i] + 1};} \\
\text{\texttt{a[i] := a[i] + 1;}} \\
\text{\texttt{(a[2] = 2)}} \\
\end{align*}
\]

Early Simplification

```
{a[i] = 0} \\
a[i] := a[i] + 1; \\
a[i] := a[i] + 1; \\
{a[2] = 2} \\
```

Deferred Simplification Alternative

- Use sequence, then array-assignment twice (from the bottom up) to get to this point:
Read-Write Example

- Suppose we wish to copy one array \( a \) into another \( b \).
- We’ll assume both arrays are the same length.
- We need to assert that they have the same elements in the same positions.

Completed Proof, using Deferred Simplification

Original program

Proposed Program

\[
\text{(i := 0)} \\
\text{while } i < \text{length(a)} \text{ do } b[i] := a[i]; i := i + 1 \text{ od} \\
\{ \forall j. (0 \leq j < \text{length(b)}) \Rightarrow b[j] = a[j] \} 
\]

One would hope for a short proof, but ...

Invariants Part of Loop Body

Program with Invariant (for JAPE)

WHERE DISTINCT n,a,b,i IS

\[
\text{(i := 0)} \\
\text{while } i < \text{length(a)} \text{ do b[i] := a[i]; i := i + 1 od} \\
\{ \forall j. (0 \leq j < \text{length(b)}) \Rightarrow b[j] = a[j] \}
\]

Assumption and Expectation

\[
\{ n = \text{length(a)} \land n = \text{length(b)} \land 0 \leq n \} \\
\ldots \text{array-copy program} \ldots \\
\{ \forall j. (0 \leq j < n) \Rightarrow b[j] = a[j] \} \\
\text{assuming indices run 0, ..., n-1} 
\]
Exercises

- Provide assumption and expectation specifications for a sorting program, then prove its total correctness.

- Work through the minimal sum-section (Huth&Ryan, Example 4.19). Note the implications for algorithmic efficiency (linear for the given method vs. $O(n^3)$ for the naïve method).