Recall the **while** rule

- In order to use the while rule in JAPE, it is necessary to supply an invariant, \( I \).

\[
\{ I \land P \} \ \mathcal{S} \ \{ I \}
\]

\[
\{ I \} \ \text{while}( \ P \ ) \ \mathcal{S} \ \{ I \land \neg P \}
\]

Inferring invariants

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

Using the **while** rule

- Before using Jape’s while rule, this setup is decomposed using Jape’s **Ntuple** rule.

\[
\begin{array}{l}
(1 \land k=0 \land j=0; k=0) \land (2; k=0) \land (3; k=0) \\
\hline
(1; 0 \land k=0 \land j=0) \land (2; 0) \land (3; 0) \\
\hline
(1; 0 \land k+1=0 \land j=0) \land (2; k=0) \land (3; k=0) \\
\end{array}
\]

A proof of a simple program
Subtleties about loop invariants

- Can the following be proved?

\[ y = 0; \ i = 0; \ n = 0; \ j = 0 \]
while \( i < n \) do
    \( y = y + j + j + 2; \ i = i + 1 \)
\od y = n \times n

Provided:
DISTINCT \( i, j, y \)

The loop invariant is not strong enough to enable induction

- This is more likely provable.

\[ y = 0; \ i = 0; \ n = 0; \ j = 0 \]
while \( i < n \) do
    \( y = y + j + j + 2; \ i = i + 1 \)
\od y = n \times n

Provided:
DISTINCT \( i, j, y \)

What is \( _M \) for termination?

\[ y = 0; \ i = 0; \ n = 0; \ j = 0 \]
while \( i < n \) do
    \( y = y + j + j + 2; \ i = i + 1 \)
\od y = n \times n

Provided:
DISTINCT \( i, j, y \)

A Completed Proof (lines 1-24 of 51)

- Arrays present extra challenges and interesting issues.
- A useful dichotomy:
  - Programs with read-only arrays
  - Programs with modifiable arrays

The Completed Proof (lines 25-51 of 51)

Other assignments in loop body

- Provides a useful dichotomy for analyzing array programs.
- Helps in identifying cases where read-only versus modifiable arrays impact the proof.

Verifying Array Programs
Array Mathematics

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

- Because several indices can have the same value, arrays are more susceptible to variable aliasing, e.g.
  \[i := 5; j = 6-1; a[j] = a[i]+1\]

Read-Only Array Example

This program sets \(j\) to the last index \(i\) such that \(a[i] = 0\). The array is assumed to be indexed \(0..n-1\).
If there is no such value, it leaves \(j\) at its initial value \(n\). (n ≥ 0 ∧ length(a)=n)

\[
\begin{align*}
  &i := 0; \\
  &j := n; \\
  &\text{while } i < n \text{ do} \\
  &\quad \text{if } a[i] = 0 \text{ then } j := i \\
  &\quad \text{else skip} \\
  &\quad \text{fi} \\
  &\quad i := i+1 \\
  &\text{od} \\
\end{align*}
\]

\{j < n → a[j] = 0\}
Read-Only Example (lines 48-74)

Quantifiers

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

Quantifier Example

- Look at part of the invariant here.
- Note that the lower bound on $x$ is a function of the index $i$.
- This is important, because it says that the element such that $a[x] = 0$ is yet to be found.
- We need this invariant to prove termination.
- The loop test will stop when $a[i] = 0$.
- The expansion order is tricky.

Quantifier Example Proved

- Things go pretty routinely, until this ...

3-elimination to the rescue

How do we get $i+1 < \text{length}(a)$?
Now case analysis

Setting up for case analysis

Strategy: ψ-Elimination

Upper branch: ∃-introduction

Upper branch closure

Lower branch

Collapsed detail
Closure of lower branch

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 Notation: \( a \oplus i \rightarrow v \) is the array that is like \( a \) except that the value of \( a[i] \) is \( v \).
- So \( (a \oplus i \rightarrow v)[i] = v \), and
  \( (a \oplus i \rightarrow v)[j] = a[j] \) if \( j \neq i \).

Jape’s Indexing rules
- \( (a \oplus i \rightarrow v)[i] = v \), and
  \( (a \oplus i \rightarrow v)[j] = a[j] \) if \( j \neq i \).
- Two rules below capture the two cases preceding.
  - The first rule simplifies an array modification expression when the index of the new array is provably the same as the index to which assignment was done.
  - The second rule simplifies in the case of a different index.
- The buttons indicate the direction of substitution.

Array Bounds Guarantees
- If an array index value is used in an assumption, the same index value can be used later on without requiring a bounds check.
- Sub-formula select a hypothesis using the desired index.

Using the Array Rule
- Make sure the entire array sub-expression is sub-formula selected.
- It should match the form in the rule in the menu:

Here we identify:
- \( A \) with \( a \)
- \( E \) with \( i \)
- \( F \) with \( a[i]+1 \)
- \( G \) with \( [i] \) (so \( E = G \)).
Using the Equality Rule

- Two selections and a sub-formula selection are needed:
  - Selection an equality hypothesis and a goal.
  - Sub-formula select an instance of the LHS of the equality.

Result of the Equality Rule

The top simple equation can be justified by "obviously".

Summary: Jape proof with array modification

How to get these rules to work in the GUI (It isn't so obvious.)

- Looking at the 2nd provided array program example, we use sequence, then array-assignment twice (from the bottom up) to get to this point:

How to use GUI (continued)

- The top line is pure logic, so we expand using →Introduction and ∧Introduction:

How to use GUI (continued)

- We then conclude the two array index bounds (lines 4, 5) by ∧ Elimination, giving:
How to use GUI (continued)

- We are left with a nested array-modification expression. Carefully sub-formula select the outer array-modification and apply the rule shown (since we have \( a[i] = \ldots[a_i] \)). Do not have anything else (such as a goal) selected.

- Repeat the preceding process on the new formula:

Consecutive Array Modification

original program
The following are more detail on an earlier example.

\[ \exists x (0 \leq x \land \text{length}(x) \land a(x) = 0) \]
\[ \{ i = i_1 \land \text{length}(i) = 0 \} \]

- Look at part of the invariant here.
- Note that the lower bound on x is a function of the index i.
- This is important, because it says that the element such that \( a[x] = 0 \) is yet to be found.
- We need this invariant to prove termination.
- The loop test will stop when \( a[i] = 0 \).
- The expansion order is tricky.

Key Step #1: Split \( i \leq i_1 \)

Then use \( \lor \) Elimination.

Key Step #2: Aim for a contradiction in the \( i = i_1 \) case.

Now introduce a backward \( \neg \) Elimination.

Key Step #2, continued:

Key Step #2, continued:

Key Step #2, continued: unify
Key Step #2, continued: use comparison menu to justify $\neg[a[i] = 0]$

Key Step #2, concluded: substitute to justify $a[i] = 0$

Status following key step #2

Completed Proof (lines 1-15)

Completed Proof (lines 16-39)

Completed Proof (lines 40-60)