(Imperative) Program Logic Part 2

Robert Keller
February 2011

Recall the **while** rule

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

$$\{I \wedge P\}\ S\ \{I\}$$

$$\{I\}\ \textbf{while(P) S}\ \ \{I \wedge \neg P\}$$

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

- In JAPE, an assertion **implying** the invariant (by using the consequent(L) rule) is included as an assertion before the while, and also doubly serves as a post-condition for the preceding statement.

(What does this code do?)

...
{i=Ki∧j=Kj∧i≥0}(k:=0)
1: {i≥0∧k+i×j=Ki×Kj}
while i≠0 do k:=k+j;i:=i−1 od
{k=Ki×Kj}

Should imply the invariant

Invariant and negation of test should imply this.

Using the **while** rule

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

1: {i=Ki∧j=Kj∧i≥0}(k:=0)
{i≥0∧k+i×j=Ki×Kj}
...

Prove using assignment rule

{i≥0∧k+i×j=Ki×Kj}
2: while i≠0 do k:=k+j;i:=i−1 od
{k=Ki×Kj}

Prove using while rule

{i=Ki∧j=Kj∧i≥0}(k:=0)
3: {i≥0∧k+i×j=Ki×Kj}
while i≠0 do k:=k+j;i:=i−1 od
{k=Ki×Kj}

Ntuple 1,2

Ntuple rule used

A proof of a simple program

1:	i=10∧i>0	assumption
2:	i−1=10	obviously
3:	i=10∧i>0→i−1=10	→ intro 1-2
4:	{i−1=10}(i:=i−1){i=10}	variable-assignment
5:	{i=10∧i>0}(i:=i−1){i=10}	consequence(L) 3,4
6:	i=10∧i>0	assumption
7:	i>0	∧ elim 6
8:	i=10∧i>0→i>0	→ intro 6-7
9:	integer Km	assumption
10:	i=10∧i>0∧i=Km	assumption
11:	i−1<Km	obviously
12:	i=10∧i>0∧i=Km→i−1<Km	→ intro 10-11
13:	{i−1<Km}(i:=i−1){i<Km}	variable-assignment
14:	{i=10∧i>0∧i=Km}(i:=i−1){i<Km}	consequence(L) 12,13
15:	{i=10}while i>0 do i:=i−1 od{i=10∧¬(i>0)}	while 5,8,9-14
16:	i=10∧¬(i>0)→i=0	obviously
17:	{i=10}(while i>0 do i:=i−1 od){i=0}	consequence(R) 15,16

Subtleties about loop invariants

- Can the following be proved?

```
...
1: {y=0∧i=0∧n≥0}(j:=1){y=i×i∧i≤n∧i≥0}
   while i<n do y:=y+j;j:=j+2;i:=i+1 od{y=n×n}
```

Provided:
DISTINCT i, j, n, y

The loop invariant is not strong enough to enable induction

- This is more likely provable.

```
...
1: {y=0∧i=0∧n≥0}(j:=1){y=i×i∧i≤n∧i≥0∧j=2×i+1}
   while i<n do y:=y+j;j:=j+2;i:=i+1 od{y=n×n}
```

Provided:
DISTINCT i, j, n, y

- What is _M for termination?

```
...
1: {y=0∧i=0∧n≥0}(j:=1){y=i×i∧i≤n∧i≥0∧j=2×i+1}
   while i<n do y:=y+j;j:=j+2;i:=i+1 od{y=n×n}
```

Provided:
DISTINCT i, j, n, y

A Completed Proof (lines 1-24 of 51)

Initialization

1:	y=0∧i=0∧n≥0	assumption
2:	y=0	∧ elim 1
3:	i=0	∧ elim 1
4:	n≥0	∧ elim 1
5:	y=i×i	obviously, from 3,2
6:	i≤n	obviously, from 4,3
7:	i≥0	obviously, from 3
8:	1=2×i+1	obviously, from 3
9:	y=i×i∧i≤n∧i≥0∧1=2×i+1	∧ intro 5,6,7,8
10:	y=0∧i=0∧n≥0→y=i×i∧i≤n∧i≥0∧1=2×i+1	→ intro 1-9
11:	{y=i×i∧i≤n∧i≥0∧1=2×i+1}(j:=1){y=i×i∧i≤n∧i≥0∧j=2×i+1}	variable-assignment

First assignment in loop body

12:	{y=0∧i=0∧n≥0}(j:=1){y=i×i∧i≤n∧i≥0∧j=2×i+1}	consequence(L) 10,11
13:	y=i×i∧i≤n∧i≥0∧j=2×i+1∧i<n	assumption
14:	y=i×i	∧ elim 13
15:	i≥0	∧ elim 13
16:	j=2×i+1	∧ elim 13
17:	i<n	∧ elim 13
18:	y+j=(i+1)×(i+1)	obviously, from 16,14
19:	i+1≤n	obviously, from 17
20:	i+1≥0	obviously, from 15
21:	j+2=2×(i+1)+1	obviously, from 16
22:	y+j=(i+1)×(i+1)∧i+1≤n∧i+1≥0∧j+2=2×(i+1)+1	∧ intro 18,19,20,21
23:	y=i×i∧i≤n∧i≥0∧j=2×i+1∧i<n→y+j=(i+1)×(i+1)∧i+1≤n∧i+1≥0∧j+2=2×(i+1)+1	→ intro 13-22
24:	{y+j=(i+1)×(i+1)∧i+1≤n∧i+1≥0∧j+2=2×(i+1)+1}(y:=y+j){y=(i+1)×(i+1)∧i+1≤n∧i+1≥0∧j+2=2×(i+1)+1}	variable-assignment

The Completed Proof (lines 25-51 of 51)

Other assignments in loop body

25:	{y=i×i∧i≤n∧i≥0∧j=2×i+1∧i<n}(y:=y+j){y=(i+1)×(i+1)∧i+1≤n∧i+1≥0∧j+2=2×(i+1)+1}	consequence(L) 23,24
26:	{y=(i+1)×(i+1)∧i+1≤n∧i+1≥0∧j+2=2×(i+1)+1}(j:=j+2){y=(i+1)×(i+1)∧i+1≤n∧i+1≥0∧j=2×(i+1)+1}	variable-assignment
27:	{y=(i+1)×(i+1)∧i+1≤n∧i+1≥0∧j=2×(i+1)+1}(i:=i+1){y=i×i∧i≤n∧i≥0∧j=2×i+1}	variable-assignment
28:	{y=i×i∧i≤n∧i≥0∧j=2×i+1∧i<n}(y:=y+j;j:=j+2;i:=i+1){y=i×i∧i≤n∧i≥0∧j=2×i+1}	sequence 25,26,27

Condition on _M

29:	y=i×i∧i≤n∧i≥0∧j=2×i+1∧i<n	assumption
30:	i<n	∧ elim 29
31:	n−i>0	obviously, from 30
32:	y=i×i∧i≤n∧i≥0∧j=2×i+1∧i<n→n−i>0	→ intro 29-31

Termination of loop body

33:	integer Km	assumption
34:	y=i×i∧i≤n∧i≥0∧j=2×i+1∧i<n∧n−i=Km	assumption
35:	n−i=Km	∧ elim 34
36:	n−(i+1)<Km	obviously, from 35
37:	y=i×i∧i≤n∧i≥0∧j=2×i+1∧i<n∧n−i=Km→n−(i+1)<Km	→ intro 34-36
38:	{n−(i+1)<Km}(y:=y+j){n−(i+1)<Km}	variable-assignment
39:	{y=i×i∧i≤n∧i≥0∧j=2×i+1∧i<n∧n−i=Km}(y:=y+j){n−(i+1)<Km}	consequence(L) 37,38
40:	{n−(i+1)<Km}(j:=j+2){n−(i+1)<Km}	variable-assignment
41:	{n−(i+1)<Km}(i:=i+1){n−i<Km}	variable-assignment
42:	{y=i×i∧i≤n∧i≥0∧j=2×i+1∧i<n∧n−i=Km}(y:=y+j;j:=j+2;i:=i+1){n−i<Km}	sequence 39,40,41
43:	{y=i×i∧i≤n∧i≥0∧j=2×i+1}while i<n do y:=y+j;j:=j+2;i:=i+1 od{y=i×i∧i≤n∧i≥0∧j=2×i+1∧¬(i<n)}	while 28,32,33-42

Exit consequence

44:	y=i×i∧i≤n∧i≥0∧j=2×i+1∧¬(i<n)	assumption
45:	y=i×i	∧ elim 44
46:	i≤n	∧ elim 44
47:	¬(i<n)	∧ elim 44
48:	y=n×n	obviously, from 47,46,45
49:	y=i×i∧i≤n∧i≥0∧j=2×i+1∧¬(i<n)→y=n×n	→ intro 44-48
50:	{y=i×i∧i≤n∧i≥0∧j=2×i+1}while i<n do y:=y+j;j:=j+2;i:=i+1 od{y=n×n}	consequence(R) 43,49
51:	{y=0∧i=0∧n≥0}(j:=1){y=i×i∧i≤n∧i≥0∧j=2×i+1}while i<n do y:=y+j;j:=j+2;i:=i+1 od{y=n×n}	Ntuple 12,50

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 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
- 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}

```
i := 0;
j := n;
while i < n
do
if a[i] = 0
  then j := i
  else skip
fi
i := i+1
od
```

{j < n → a[j] = 0}

What invariant do we need?

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.

. . .

assumption program invariant

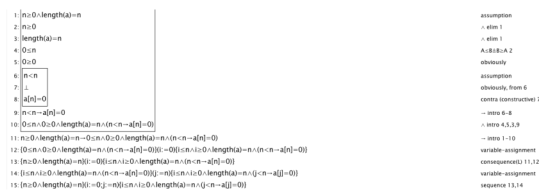
1: {n≥0∧length(a)=n}(i:=0;j:=n){i≤n∧i≥0∧length(a)=n∧(j<n→a[j]=0)}
while i<n do if a[i]=0 then j:=i else skip fi;i:=i+1 od{j<n→a[j]=0}

expectation

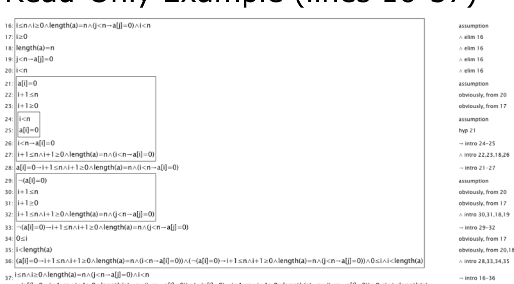
Provided:

DISTINCT a, i, j, n

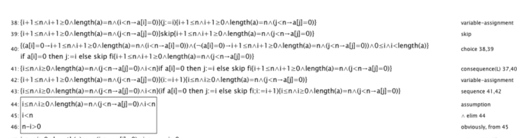
Read-Only Example (lines 1-15)



Read-Only Example (lines 16-37)



Read-Only Example (lines 38-47)



Read-Only Example (lines 48-74)

48:	integer Km	assumption
49:	i≤n∧i≥0∧length(a)=n∧(j<n→a[j]=0)∧i<n∧n−i=Km	assumption
50:	i≥0	∧ elim 49
51:	i<n	∧ elim 49
52:	n−i=Km	∧ elim 49
53:	a[i]=0	assumption
54:	n−(i+1)<Km	obviously, from 52
55:	a[i]=0→n−(i+1)<Km	→ intro 53-54
56:	¬(a[i]=0)	assumption
57:	n−(i+1)<Km	obviously, from 52
58:	¬(a[i]=0)→n−(i+1)<Km	→ intro 56-57
59:	0≤i	A≤B≜B≥A 50
60:	i<length(a)	obviously, from 51
61:	(a[i]=0→n−(i+1)<Km)∧(¬(a[i]=0)→n−(i+1)<Km)∧0≤i∧i<length(a)	∧ intro 55,58,59,60
62:	i≤n∧i≥0∧length(a)=n∧(j<n→a[j]=0)∧i<n∧n−i=Km→(a[i]=0→n−(i+1)<Km)∧(¬(a[i]=0)→n−(i+1)<Km)∧0≤i∧i<length(a)	→ intro 49-61
63:	{n−(i+1)<Km}(j:=i){n−(i+1)<Km}	variable-assignment
64:	{n−(i+1)<Km}skip{n−(i+1)<Km}	skip
65:	{(a[i]=0→n−(i+1)<Km)∧(¬(a[i]=0)→n−(i+1)<Km)∧0≤i∧i<length(a)}if a[i]=0 then j:=i else skip fi{n−(i+1)<Km}	choice 63,64
66:	{i≤n∧i≥0∧length(a)=n∧(j<n→a[j]=0)∧i<n∧n−i=Km}if a[i]=0 then j:=i else skip fi{n−(i+1)<Km}	consequence(L) 62,65
67:	{n−(i+1)<Km}(i:=i+1){n−i<Km}	variable-assignment
68:	{i≤n∧i≥0∧length(a)=n∧(j<n→a[j]=0)∧i<n∧n−i=Km}(if a[i]=0 then j:=i else skip fi;i:=i+1){n−i<Km}	sequence 66,67
69:	{i≤n∧i≥0∧length(a)=n∧(j<n→a[j]=0)}while i<n do if a[i]=0 then j:=i else skip fi;i:=i+1 od{i≤n∧i≥0∧length(a)=n∧(j<n→a[j]=0)∧¬(i<n)}	while 43,47,48-68
70:	i≤n∧i≥0∧length(a)=n∧(j<n→a[j]=0)∧¬(i<n)	assumption
71:	j<n→a[j]=0	∧ elim 70
72:	i≤n∧i≥0∧length(a)=n∧(j<n→a[j]=0)∧¬(i<n)→j<n→a[j]=0	→ intro 70-71
73:	{i≤n∧i≥0∧length(a)=n∧(j<n→a[j]=0)}while i<n do if a[i]=0 then j:=i else skip fi;i:=i+1 od{j<n→a[j]=0}	consequence(R) 69,72
74:	{n≥0∧length(a)=n}(i:=0;j:=n){i≤n∧i≥0∧length(a)=n∧(j<n→a[j]=0)}while i<n do if a[i]=0 then j:=i else skip fi;i:=i+1 od{j<n→a[j]=0}	Ntuple 15,73

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

... assumption program program

1: {∃x.(0≤x∧x<length(a)∧a[x]=0)}(i:=0)

{0≤i∧i<length(a)∧∃x.(i≤x∧x<length(a)∧a[x]=0)}while a[i]≠0 do i:=i+1 od{a[i]=0}

invariant expectation

This program assumes there is an array element having value 0. It returns an index of such a value.

This is from the JAPE "factory samples". Will the invariant do the job?

Subtleties with Array Programs

{∃x.(0≤x∧x<length(a)∧a[x]=0)}(i:=0)

{0≤i∧i<length(a)∧∃x.(i≤x∧x<length(a)∧a[x]=0)}while a[i]≠0 do i:=i+1 od{a[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.

Quantifier Example Proved

- Things go pretty routinely, until this …

3:	0≤i∧i<length(a)∧∃x.(i≤x∧x<length(a)∧a[x]=0)∧a[i]≠0
4:	0≤i
5:	i<length(a)
6:	∃x.(i≤x∧x<length(a)∧a[x]=0)
7:	a[i]≠0
	...
8:	0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)

How do we get i+1<length(a)?

∃-elimination to the rescue

3:	0≤i∧i<length(a)∧∃x.(i≤x∧x<length(a)∧a[x]=0)∧a[i]≠0
4:	0≤i
5:	i<length(a)
6:	∃x.(i≤x∧x<length(a)∧a[x]=0)
7:	a[i]≠0
8:	integer i1, i≤i1∧i1<length(a)∧a[i1]=0
	...
9:	0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
10:	0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)

Now case analysis

```
 3: 0≤i∧i<length(a)∧∃x.(i≤x∧x<length(a)∧a[x]=0)∧a[i]≠0
 4: 0≤i
 5: i<length(a)
 6: ∃x.(i≤x∧x<length(a)∧a[x]=0)
 7: a[i]≠0
 8: integer i1, i≤i1∧i1<length(a)∧a[i1]=0
 9: i≤i1
10: i1<length(a)
11: a[i1]=0
    ...
12: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
13: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
```

Setting up for case analysis

```
 3: 0≤i∧i<length(a)∧∃x.(i≤x∧x<length(a)∧a[x]=0)∧a[i]≠0
 4: 0≤i
 5: i<length(a)
 6: ∃x.(i≤x∧x<length(a)∧a[x]=0)
 7: a[i]≠0
 8: integer i1, i≤i1∧i1<length(a)∧a[i1]=0
 9: i≤i1
10: i1<length(a)
11: a[i1]=0
12: i<i1∨i=i1     from 9
    ...
13: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
14: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
```

Strategy: ∨-Elimination

```
 3: 0≤i∧i<length(a)∧∃x.(i≤x∧x<length(a)∧a[x]=0)∧a[i]≠0
 4: 0≤i
 5: i<length(a)
 6: ∃x.(i≤x∧x<length(a)∧a[x]=0)
 7: a[i]≠0
 8: integer i1, i≤i1∧i1<length(a)∧a[i1]=0
 9: i≤i1
10: i1<length(a)
11: a[i1]=0
12: i<i1∨i=i1
13: i<i1
    ...
14: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
15: i=i1
    ...
16: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
17: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
18: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
```

Upper branch: ∃-introduction

```
12: i<i1∨i=i1
13: i<i1
    ...
14: 0≤i+1
    ...
15: i+1<length(a)
    ...
16: ∃x.(i+1≤x∧x<length(a)∧a[x]=0)
17: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
```

Upper branch closure

```
 3: 0≤i∧i<length(a)∧∃x.(i≤x∧x<length(a)∧a[x]=0)∧a[i]≠0
 4: 0≤i
 5: i<length(a)
 6: ∃x.(i≤x∧x<length(a)∧a[x]=0)
 7: a[i]≠0
 8: integer i1, i≤i1∧i1<length(a)∧a[i1]=0
 9: i≤i1
10: i1<length(a)
11: a[i1]=0
12: i<i1∨i=i1
13: i<i1
14: 0≤i+1
15: i+1<length(a)
16: i+1≤i1
17: i+1≤i1∧i1<length(a)∧a[i1]=0
18: ∃x.(i+1≤x∧x<length(a)∧a[x]=0)
19: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
```

Lower branch

```
 3: 0≤i∧i<length(a)∧∃x.(i≤x∧x<length(a)∧a[x]=0)∧a[i]≠0
 4: 0≤i
 5: i<length(a)
 6: ∃x.(i≤x∧x<length(a)∧a[x]=0)
 7: a[i]≠0
 8: integer i1, i≤i1∧i1<length(a)∧a[i1]=0
 9: i≤i1
10: i1<length(a)
11: a[i1]=0
12: i<i1∨i=i1
13: i<i1
14: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
15: i=i1
    ...
16: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
```

Collapsed detail

Closure of lower branch

5:	a[i]≠0	∧ elim 3
6:	integer i1, i≤i1∧i1<length(a)∧a[i1]=0	assumptions
7:	i≤i1	∧ elim 6.2
8:	a[i1]=0	∧ elim 6.2
9:	i<i1∨i=i1	A≤B≜A<B∨A=B 7
10:	i<i1	assumption
11:	0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)	(cut)
12:	i=i1	assumption
13:	a[i]=0	equality-substitution 12,8
14:	¬(a[i]=0)	A≠B≜¬(A=B) 5
15:	⊥	¬ elim 13,14
16:	0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)	contra (constructive) 15

```
14: 0≤i∧i<length(a)∧∃x.(i≤x∧x<length(a)∧a[x]=0)∧a[i]≠0
15: 0≤i
16: ∃x.(i≤x∧x<length(a)∧a[x]=0)
17: a[i]≠0
18: integer i1, i≤i1∧i1<length(a)∧a[i1]=0
19: i≤i1
20: i1<length(a)
21: a[i1]=0
22: i<i1∨i=i1
23: i<i1
24: 0≤i+1
25: i+1<length(a)
26: i+1≤i1
27: i+1≤i1∧i1<length(a)∧a[i1]=0
28: ∃x.(i+1≤x∧x<length(a)∧a[x]=0)
29: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
30: i=i1
31: a[i]=0
32: ¬(a[i]=0)
33: ⊥
34: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
35: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
36: 0≤i+1∧i+1<length(a)∧∃x.(i+1≤x∧x<length(a)∧a[x]=0)
```

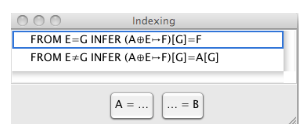
Modifiable Arrays

- 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⊕i→v is the array that is like a except that the value of a[i] is v.
- So (a⊕i→v)[i] = v, and
  (a⊕i→v)[j] = a[j] if j ≠ i.

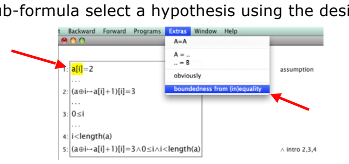
Jape's Indexing rules

- (a⊕i→v)[i] = v, and
  (a⊕i→v)[j] = a[j] if j ≠ 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.

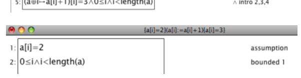


Array Bounds Guarantees

- 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.



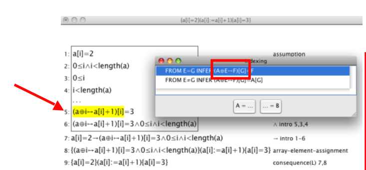
Result:



Using the Array Rule

- Make sure the entire array sub-expression is sub-formula selected.

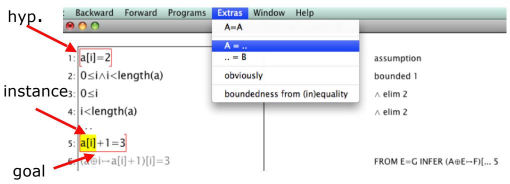
- It should match the form in the rule in the menu:



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

```
     ...
 5:  2+1=3
 6:  a[i]+1=3                                          equality-substitution 1,5
 7:  (a⊕i↦a[i]+1)[i]=3                                 FROM E=G INFER (A⊕E↦F)[G... 6
 8:  (a⊕i↦a[i]+1)[i]=3∧0≤i∧i<length(a)                 ∧ intro 7,3,4
 9: a[i]=2→(a⊕i↦a[i]+1)[i]=3∧0≤i∧i<length(a)           → intro 1-8
10: {(a⊕i↦a[i]+1)[i]=3∧0≤i∧i<length(a)}(a[i]:=a[i]+1){a[i]=3} array-element-assignment
11: {a[i]=2}(a[i]:=a[i]+1){a[i]=3}                      consequence(L) 9,10
```

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

Summary: Jape proof with array modification

```
 1:  a[i]=2                                            assumption
 2:  0≤i∧i<length(a)                                   bounded 1
 3:  0≤i                                               ∧ elim 2
 4:  i<length(a)                                       ∧ elim 2
 5:  2+1=3                                             obviously
 6:  a[i]+1=3                                          equality-substitution 1,5
 7:  (a⊕i↦a[i]+1)[i]=3                                 FROM E=G INFER (A⊕E↦F)[G]=F 6
 8:  (a⊕i↦a[i]+1)[i]=3∧0≤i∧i<length(a)                 ∧ intro 7,3,4
 9: a[i]=2→(a⊕i↦a[i]+1)[i]=3∧0≤i∧i<length(a)           → intro 1-8
10: {(a⊕i↦a[i]+1)[i]=3∧0≤i∧i<length(a)}(a[i]:=a[i]+1){a[i]=3} array-element-assignment
11: {a[i]=2}(a[i]:=a[i]+1){a[i]=3}                      consequence(L) 9,10
```

infers in-bounds from usage in 1.

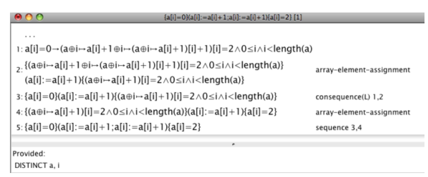
A=… rule

index rule

statement

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:

```
1:  a[i]=0                                                          assumption
    ...
2:  (a⊕i↦a[i]+1⊕i↦(a⊕i↦a[i]+1)[i]+1)[i]=2
    ...
3:  0≤i
    ...
4:  i<length(a)
5:  (a⊕i↦a[i]+1⊕i↦(a⊕i↦a[i]+1)[i]+1)[i]=2∧0≤i∧i<length(a)           ∧ intro 2,3,4
6: a[i]=0→(a⊕i↦a[i]+1⊕i↦(a⊕i↦a[i]+1)[i]+1)[i]=2∧0≤i∧i<length(a)     → intro 1-5
```

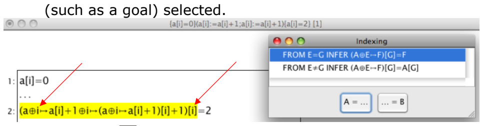
How to use GUI (continued)

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

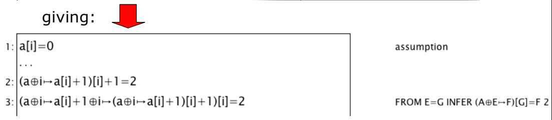
```
1:  a[i]=0                                                          assumption
    ...
2:  (a⊕i↦a[i]+1⊕i↦(a⊕i↦a[i]+1)[i]+1)[i]=2
3:  0≤i∧i<length(a)                                                 bounded 1
4:  0≤i                                                             ∧ elim 3
5:  i<length(a)                                                     ∧ elim 3
6:  (a⊕i↦a[i]+1⊕i↦(a⊕i↦a[i]+1)[i]+1)[i]=2∧0≤i∧i<length(a)           ∧ intro 2,4,5
```

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→ …[i]). Do not have anything else (such as a goal) selected.

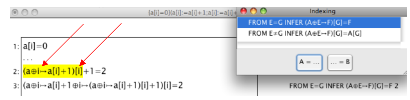


giving:



How to use GUI (continued)

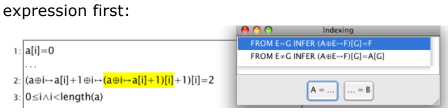
- Repeat the preceding process on the new formula:



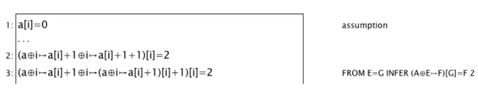
giving:

How to use GUI (continued)

- Alternatively we could have selected the *inner* modification expression first:

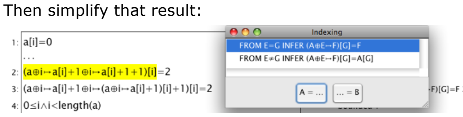


giving:

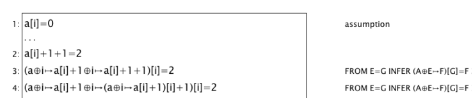


How to use GUI (continued)

- (Note that this is different from two slides ago). Then simplify that result:

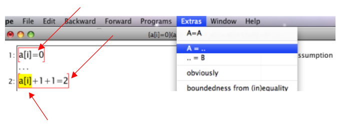


giving (as before):

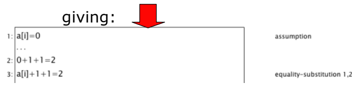


How to use GUI (continued)

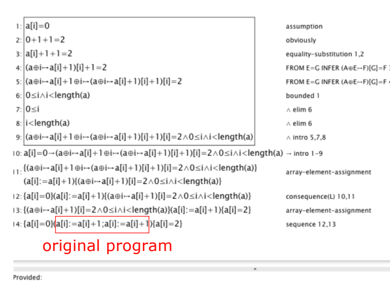
- Use plain equality substitution to simplify the new goal. Note that both the goal and the equation defining the substitution are selected, and the sub-formula for which substitution is to occur is sub-formula selected (3 selections).



giving:



Consecutive Array Modification



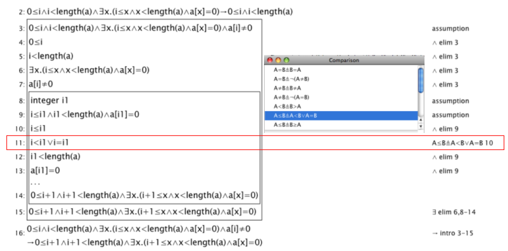
The following are more detail on an earlier example.

$\{\exists x.(0\le x\wedge x<length(a)\wedge a[x]=0)\}(i:=0)$

$\{0\le i\wedge i<length(a)\wedge \boxed{\exists x.(i\le x\wedge x<length(a)\wedge a[x]=0)}\}$while $a[i]\ne 0$ do $i:=i+1$ od$\{a[i]=0\}$

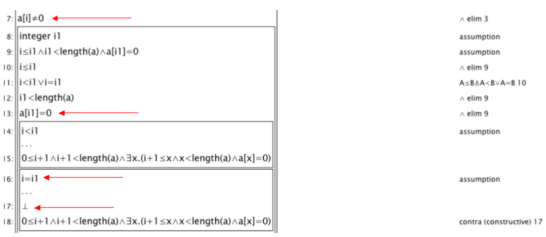
- Look at part of the invariant here.
- Note that the lower bound on x is a function of the index i.
- This is important, because it says that the element such that **a[x] = 0 is yet to be found**.
- We need this invariant to prove termination.
- The loop test will stop when a[i] = 0.
- The expansion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

Key Step #1: Split i $\le$ i1



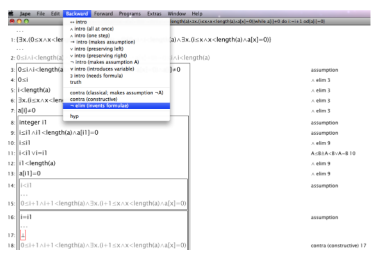
Then use ∨Elimination.

Key Step #2:
Aim for a contradiction in the i = i1 case.

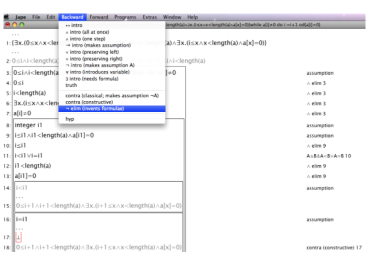


Now introduce a backward ¬Elimination.

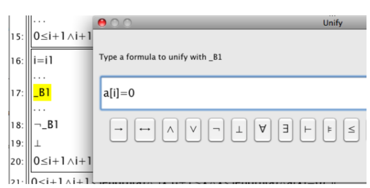
Key Step #2, continued:



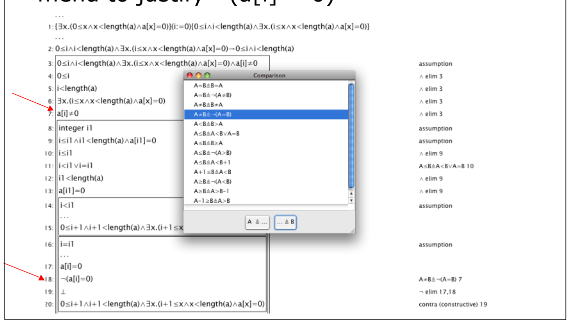
Key Step #2, continued:



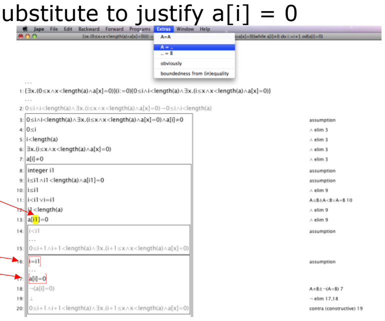
Key Step #2, continued: unify



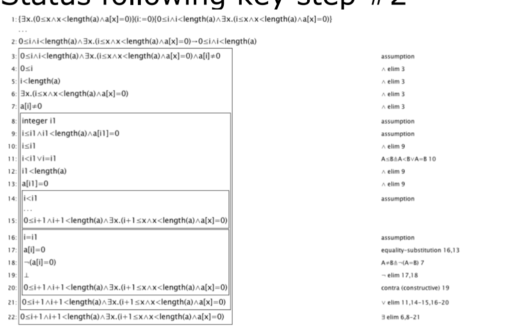
Key Step #2, continued: use comparison menu to justify ¬(a[i] = 0)



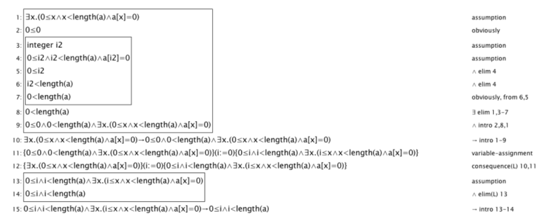
Key Step #2, concluded: substitute to justify a[i] = 0



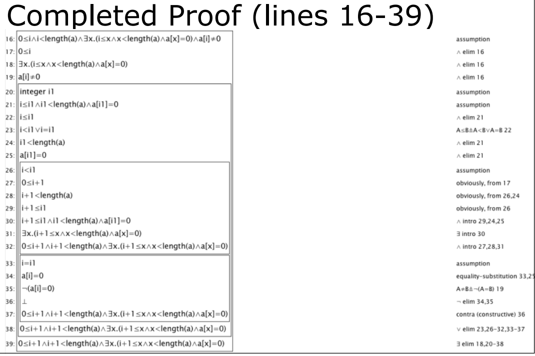
Status following key step #2



Completed Proof (lines 1-15)



Completed Proof (lines 16-39)



Completed Proof (lines 40-60)