Loop Verification with Invariants and Contracts [VMCAI 2022]

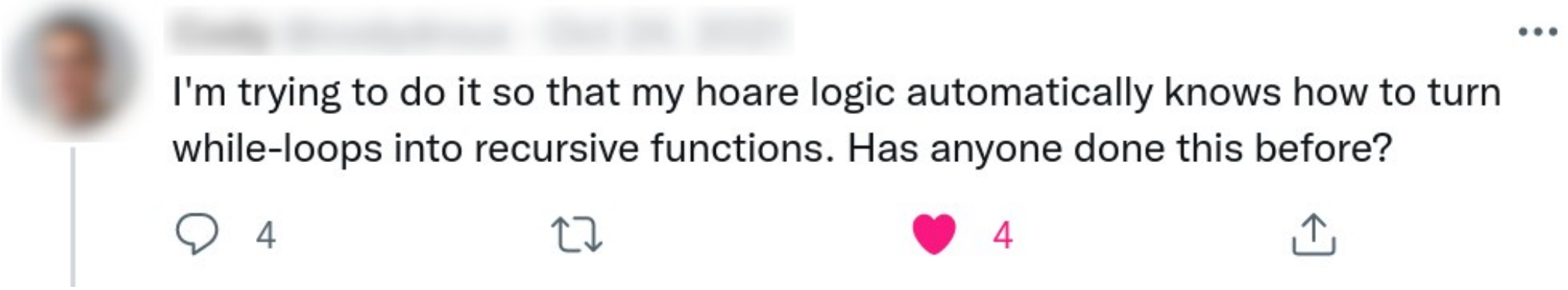
Gidon Ernst, LMU Munich gidon.ernst@lmu.de

November 10, 2022, Dagstuhl Seminar
Principles of Contract Languages

https://www.sosy-lab.org

“ recursive algorithms
are easier to prove correct
than iterative ones ” --- folklore

“ recursive algorithms
are easier to prove correct
than iterative ones ” --- folklore



“ recursive algorithms
are easier to prove correct
than iterative ones ” --- folklore

I'm trying to do it so that my hoare logic automatically knows how to turn while-loops into recursive functions. Has anyone done this before?

Related: Hehner’s specified blocks, Tuerk’s local while rules, HOLfoot, VeriFast, KeY, Beringer’s “nonstandard” rule, general-purpose ITPs, software model checkers

Loop Contracts generalize Proofs with Invariants

$$\frac{\{\ t(x)\ \wedge\ I(x)\ \}\ \texttt{body}\ \{\ I(x)\ \}}{\{\ I(x)\ \}\ \texttt{while t do B}\ \{\ \neg t(x)\ \wedge\ I(x)\ \}}$$

invariant

“canonical” conclusion

Loop Contracts generalize Proofs with Invariants

$$\frac{\{\ t(x)\ \wedge\ I(x)\ \}\ \text{body}\ \{\ I(x)\ \}}{\{\ I(x)\ \}\ \text{while t do B}\ \{\ \neg t(x)\ \wedge\ I(x)\ \}}$$

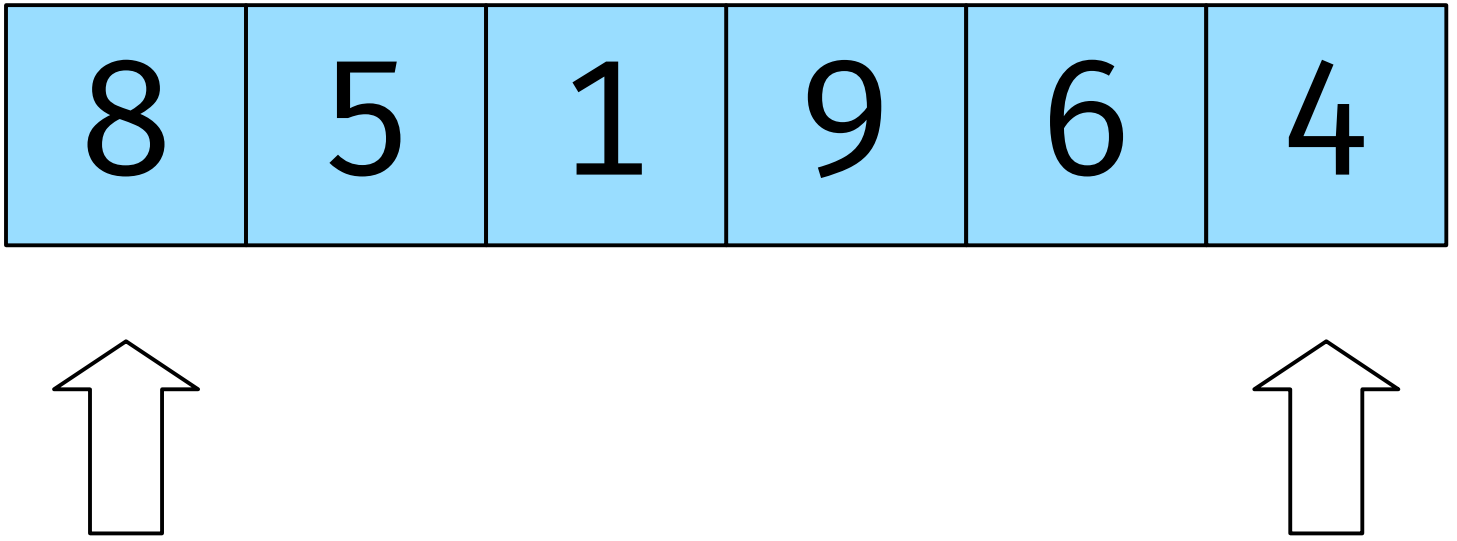
invariant

"canonical" conclusion

$$\frac{\text{(treat loop as tail-recursive procedure)}}{\{\ I(x)\ \wedge\ x = x_0\ \}\ \text{while t do B}\ \{\ R(x_0,\ x)\ \}}$$

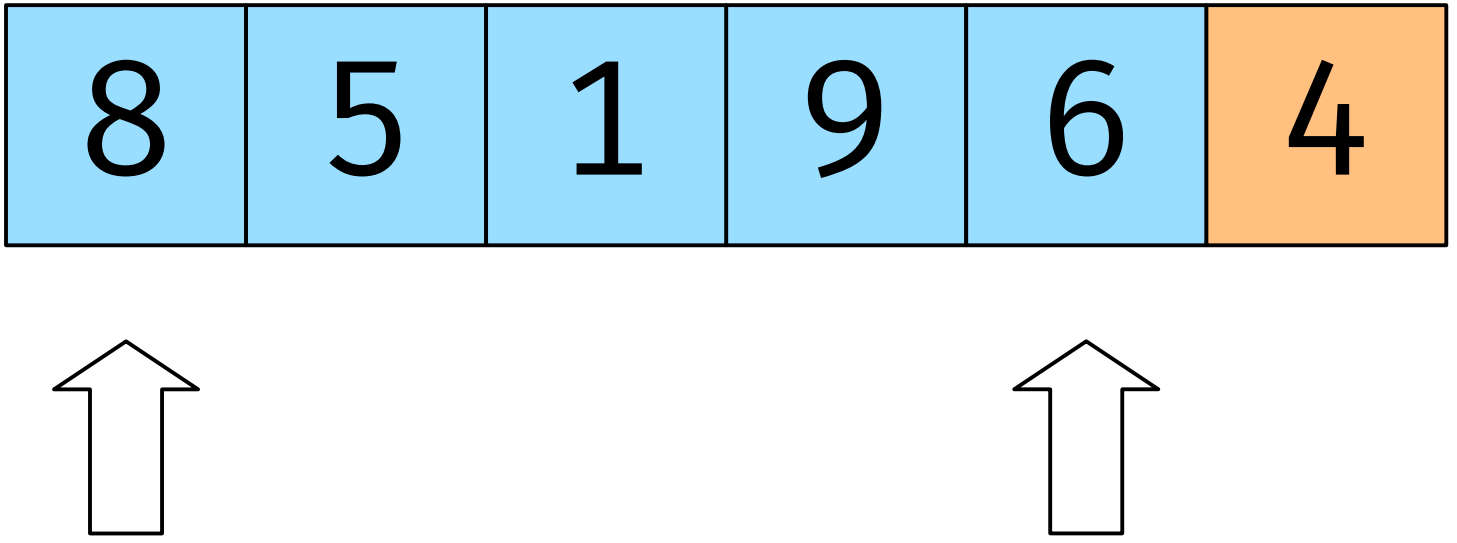
contract = invariant + **relational** summary

Find max by elimination (VerifyThis 2011)



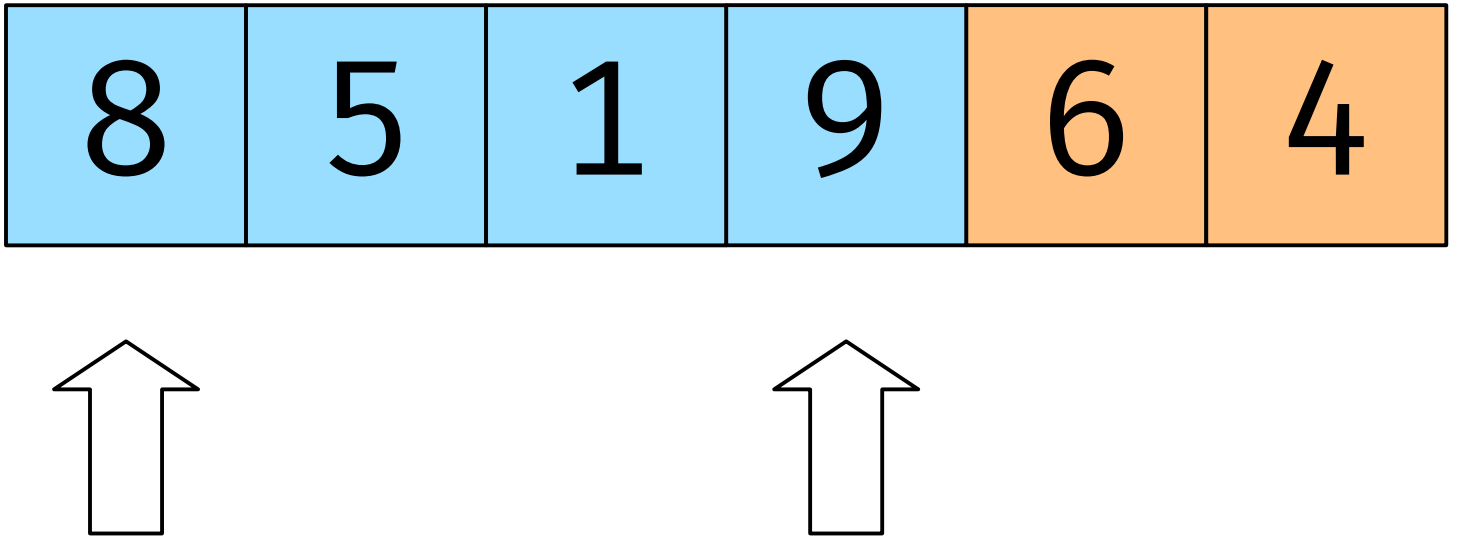
Find max by elimination **(VerifyThis 2011)**

eliminate 4 because $4 \leqslant 8$



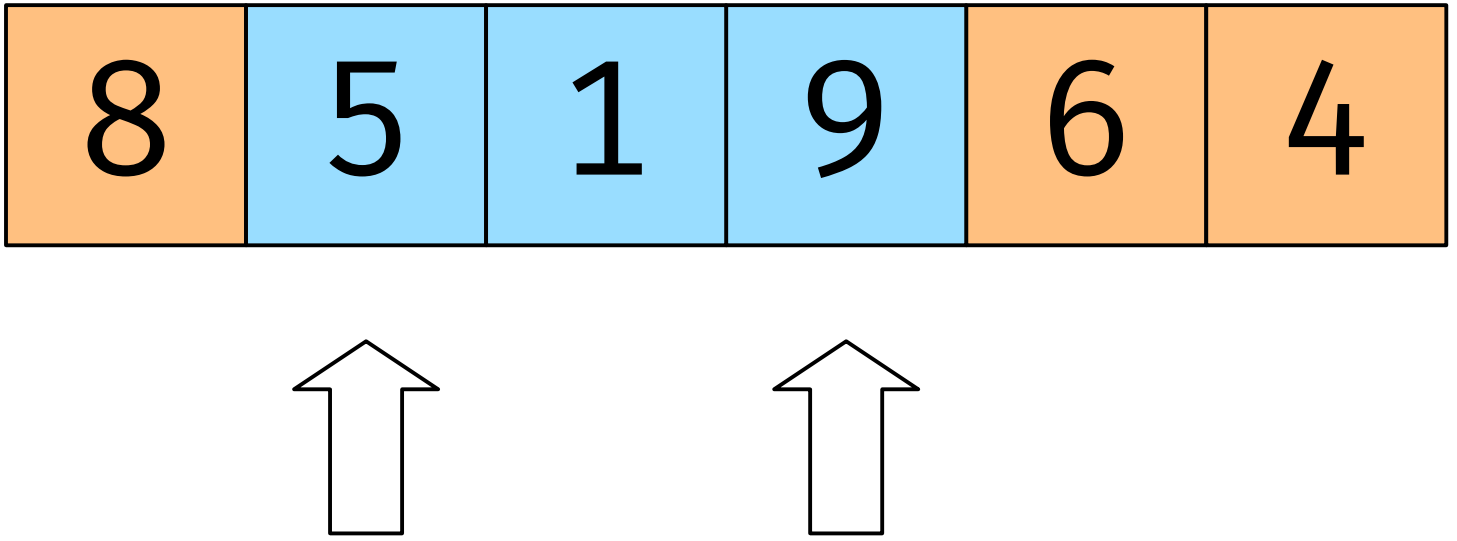
Find max by elimination **(VerifyThis 2011)**

eliminate 6 because $6 \leqslant 8$

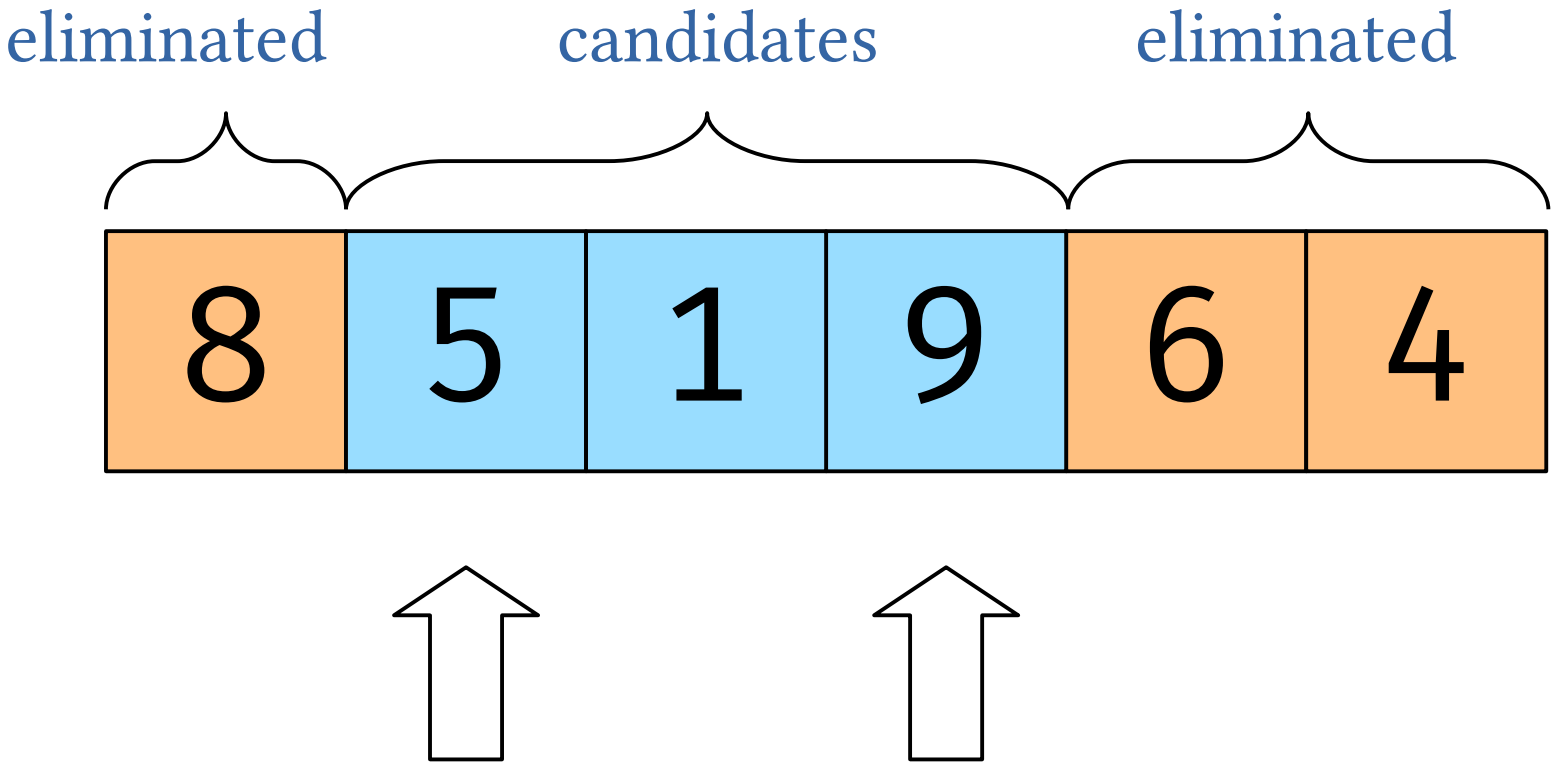


Find max by elimination **(VerifyThis 2011)**

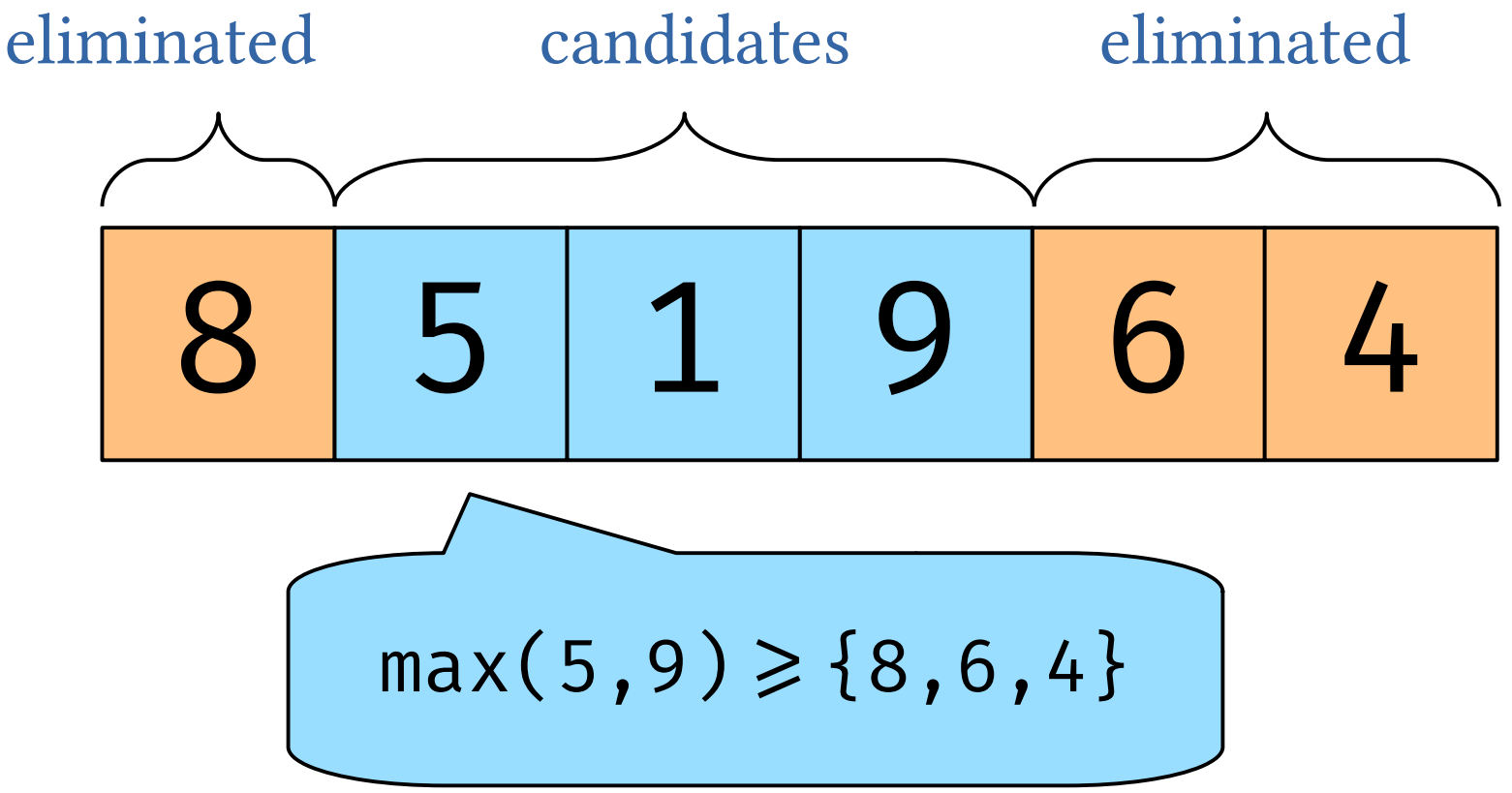
eliminate 8 because $8 \leqslant 9$



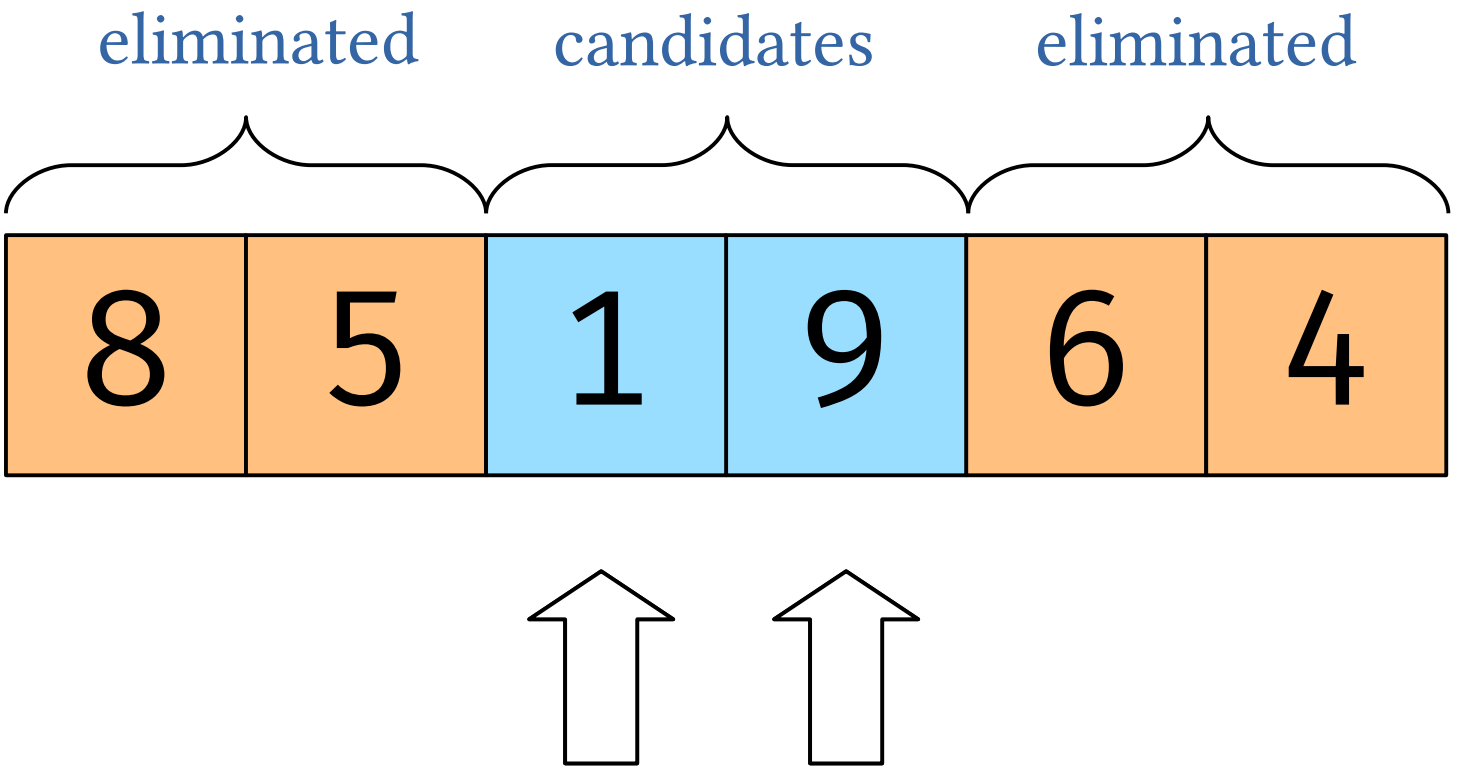
Find max by elimination (VerifyThis 2011)



Find max by elimination (VerifyThis 2011)

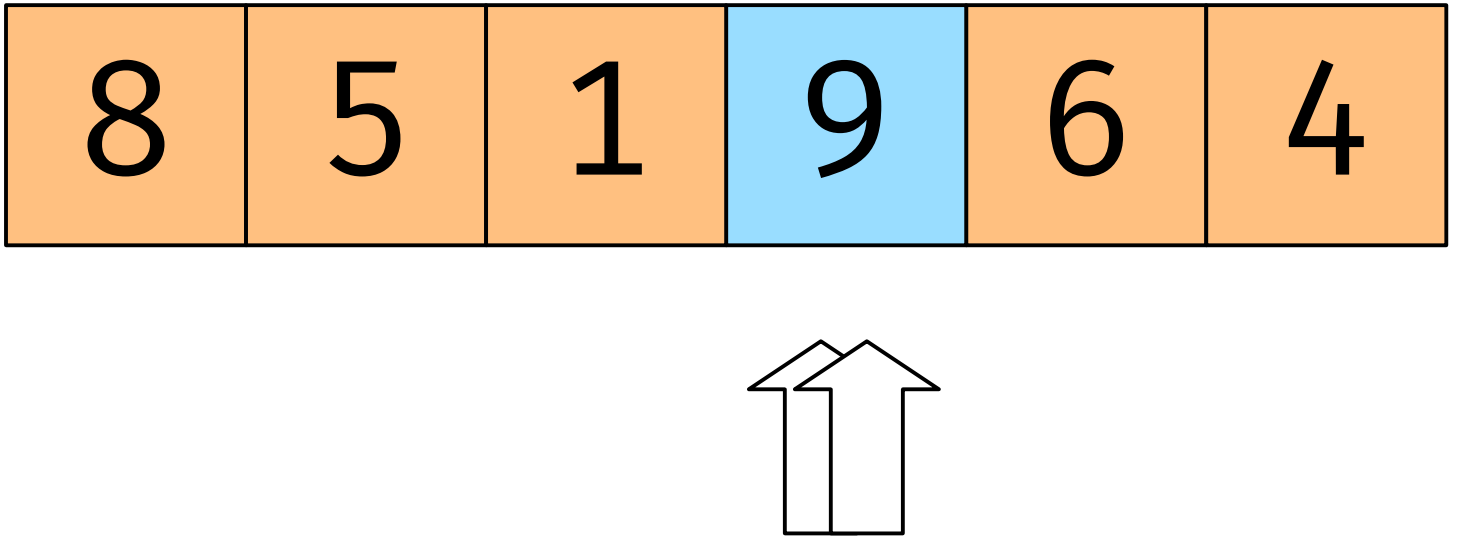


Find max by elimination (VerifyThis 2011)

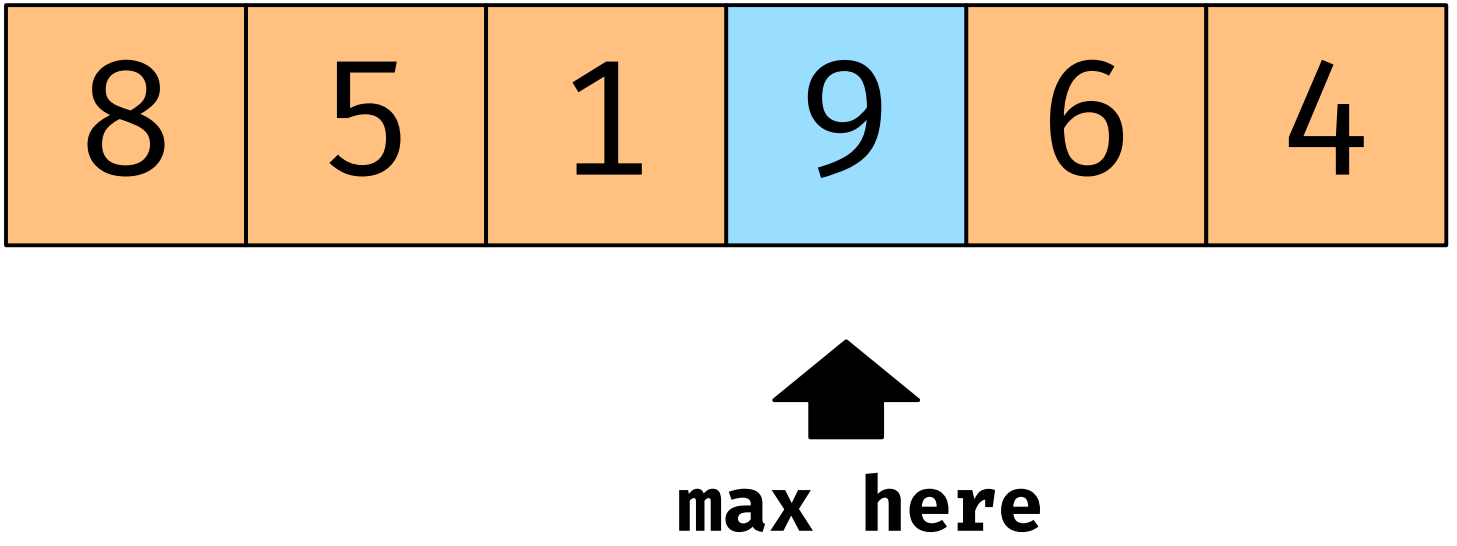


Find max by elimination **(VerifyThis 2011)**

eliminate 5 and 1 $\leqslant$ 9



Find max by elimination (VerifyThis 2011)



Recursive algorithm

```
int findMax(int a[], int l, int r)
  requires ???
  ensures  ???


  if l == r then return l
  else if a[l] ⩽ a[r] then return findMax(a, l+1, r)
  else if a[l] ⩾ a[r] then return findMax(a, l, r-1)
```

Recursive algorithm

```
int findMax(int a[], int l, int r)
  requires 0 ⩽ l ⩽ r < a.length
  ensures  a[\result] == max(a[l..r+1])


  if l == r then return l
  else if a[l] ⩽ a[r] then return findMax(a, l+1, r)
  else if a[l] ⩾ a[r] then return findMax(a, l, r-1)
```

relation between parameters and return value

Recursive algorithm

```
int findMax(int a[], int l, int r)
  requires 0 ≤ l ≤ r < a.length
  ensures  a[\result] == max(a[l..r+1])
```

$$\forall\ i.\quad l \leqslant i < r+1 \implies a[i] \leqslant a[\result]$$

```
  if l == r then return l
  else if a[l] ⩽ a[r] then return findMax(a, l+1, r)
  else if a[l] ⩾ a[r] then return findMax(a, l, r-1)
```

Recursive algorithm

```
int findMax(int a[], int l, int r)
  requires 0 ≤ l ≤ r < a.length
  ensures  a[\result] == max(a[l..r+1])

  ∀ i.  l ≤ i < r+1  ⟹  a[i] ≤ a[\result]

  if l == r then return l
  else if a[l] ≤ a[r] then return findMax(a, l+1, r)
  else if a[l] ≥ a[r] then return findMax(a, l, r-1)
```

sufficient for
- automatic proof
- calling context

Iterative algorithm

```
l := 0,   r := a.length − 1
```

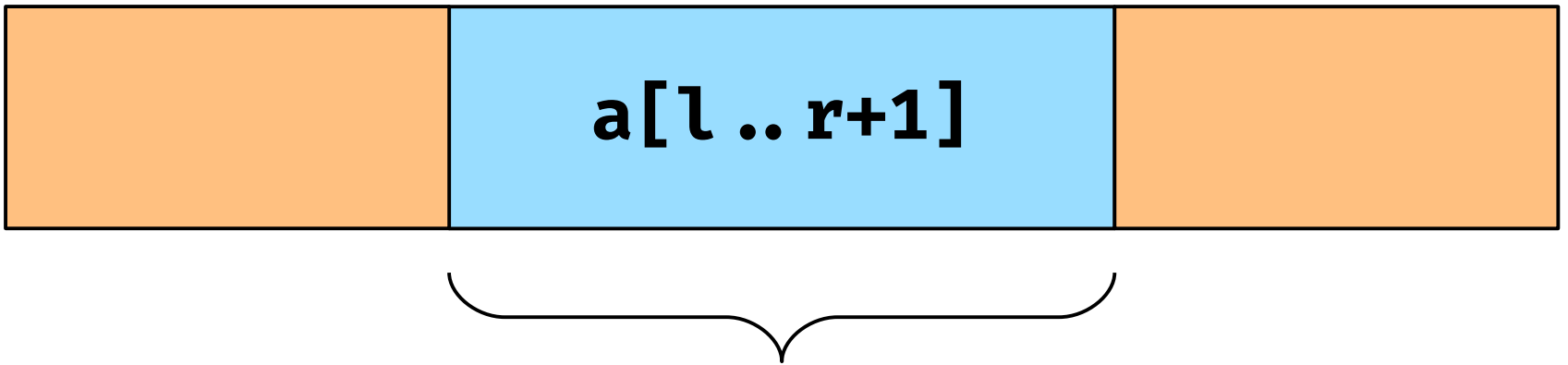
initial state
(omitted previously)

```
while true do
    invariant ???

  if l = r then return l
  else if a[l] ⩽ a[r] then l := l+1
  else if a[l] ⩾ a[r] then r := r-1
```

in-place update
(cf. tail-recursion
elimination)

Invariant for max by elimination



contains overall max

(see VerifyThis 2011 report for a variety of formalizations)

Iterative algorithm

```
l := 0,  r := a.length - 1

while true do
  invariant ∃ k.      0 ⩽ l ⩽ k ⩽ r < a.length
                    ∧ a[k] == max(a[0..a.length])
  if l == r then return l
  else if a[l] ⩽ a[r] then l := l+1
  else if a[l] ⩾ a[r] then r := r-1
```

sufficient for
- automatic proof
- postcondition (omitted)

“Invariant” proof of the recursive algorithm

```
int findMax(int a[], int l, int r)
  requires ∃ k.  0 ⩽ l ⩽ k ⩽ r < a.length
                   ∧ a[k] == max(a[0..a.length])
  ensures  ∃ k.  0 ⩽ \result ⩽ k ⩽ \result < a.length
                   ∧ a[k] == max(a[0..a.length])
```

"Invariant" proof of the recursive algorithm

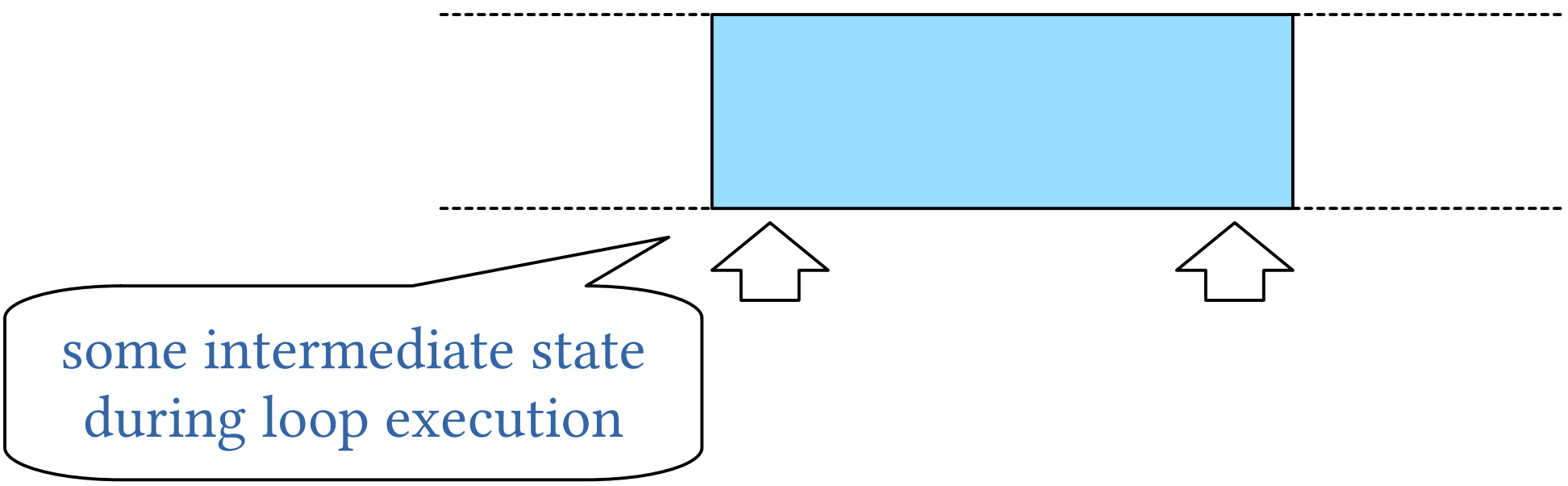
```
int findMax(int a[], int l, int r)
  requires ∃ k.  0 ⩽ l ⩽ k ⩽ r < a.length
                   ∧ a[k] == max(a[0..a.length])
  ensures  ∃ k.  0 ⩽ \result ⩽ k ⩽ \result < a.length
                   ∧ a[k] == max(a[0..a.length])
```

Self-imposed limitation:

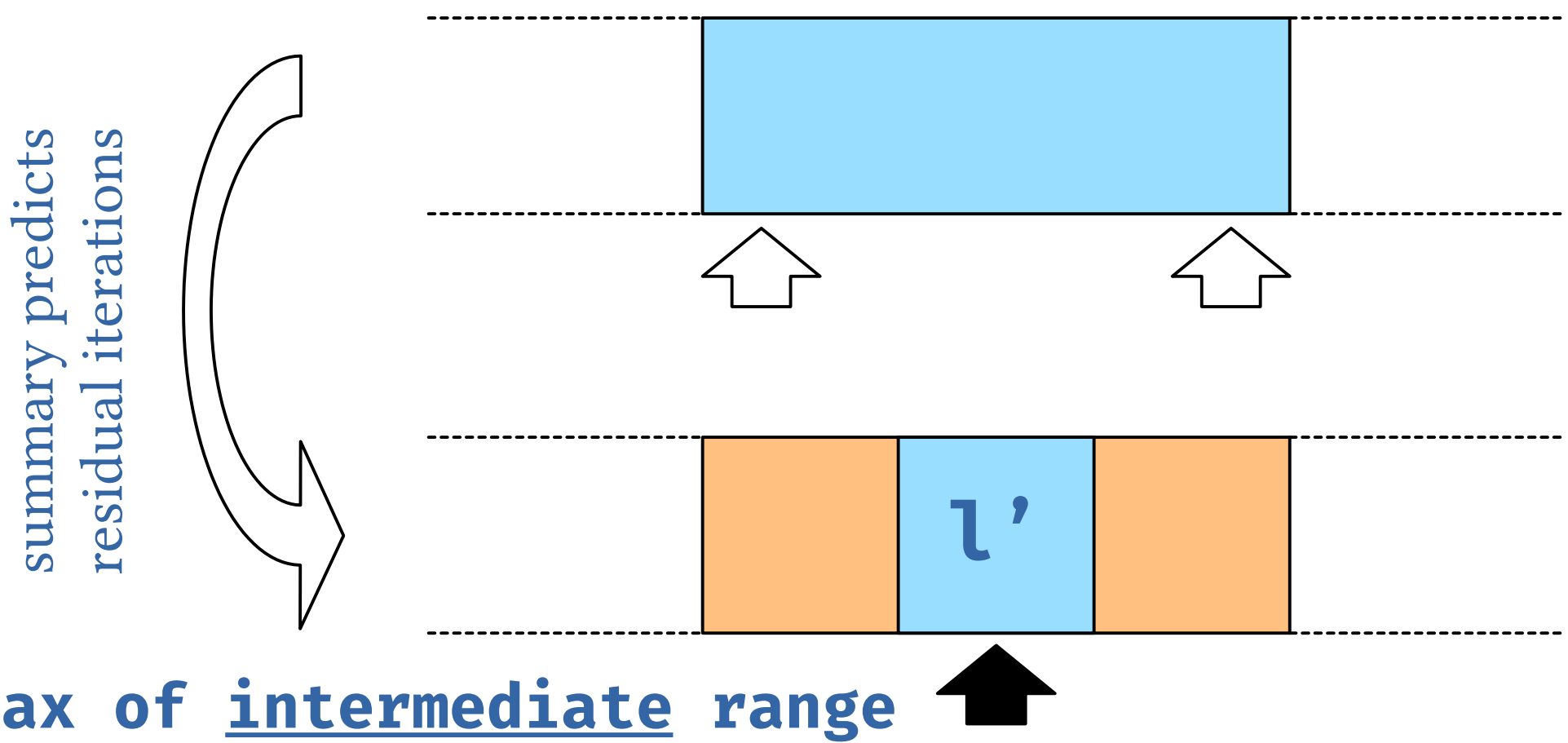
loops with invariants $\Longleftrightarrow$

tail recursion where pre-/postcondition are the same

Relational summary: `a[final(l)] == max(a[l .. r+1])`



Relational summary: `a[final(l)] = max(a[l .. r+1])`



Loop Contract for iterative version

```
l := 0,  r := a.length - 1

while true do
  requires 0 ⩽ l ⩽ r < a.length
  ensures  a[final(l)] == max(a[l..r+1])
assert  a[l] == max(a[0..a.length])
```

syntactic reference to final value of `l`

loop summary very similar to postcondition!

Demo: Cuvée

https://github.com/gernst/cuvee

Loop Contracts **[Hehner, Tuerk, …]**

1. … generalize proofs with invariants
2. … sometimes capture proofs quite naturally

Invariant

- describes work done so far
- reasons about *partial* results

Summary

- describes residual work
- reasons about *complete* results but for intermediate states

Loop Contracts [Hehner, Tuerk, ...]

1. ... generalize proofs with invariants
2. ... sometimes capture proofs quite naturally

This work: explain loop contracts

- *simple* formalization of verification conditions
- clarify correspondence between invariants and summaries
- tool construction guidelines
- examples with text-book verification challenges

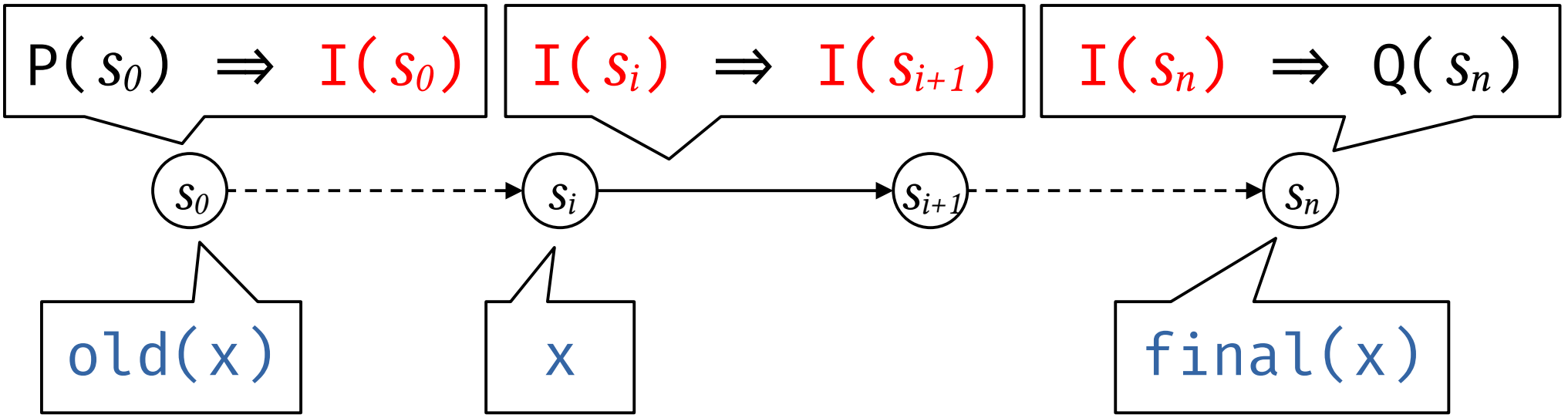
Loop Contracts [Hehner, Tuerk, ...]

1. ... generalize proofs with invariants
2. ... sometimes capture proofs quite naturally

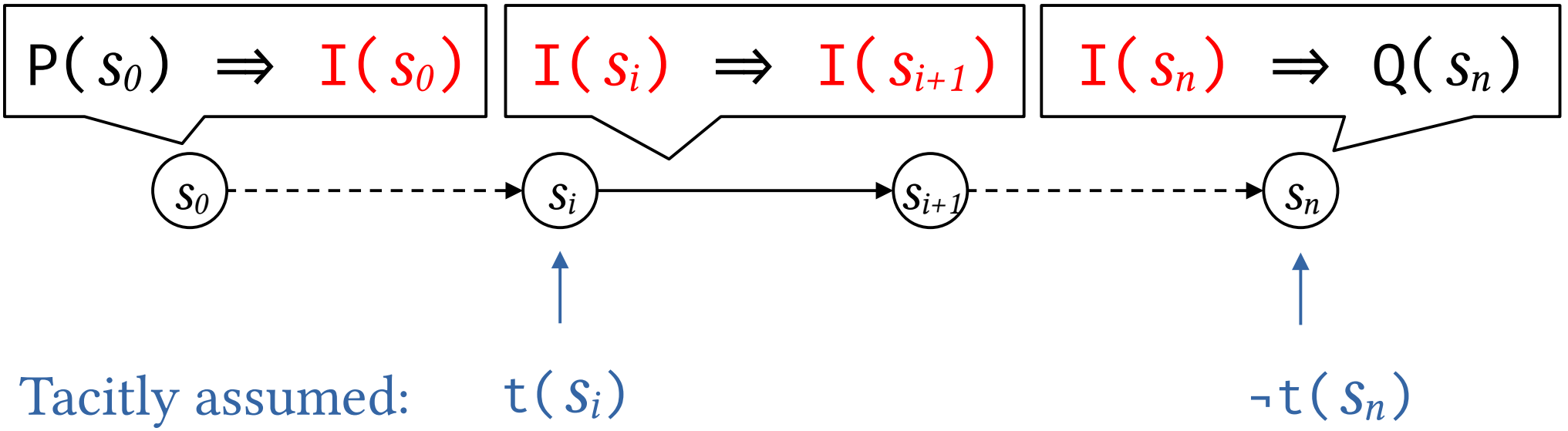
This work: explain loop contracts

- *simple* formalization of verification conditions
- clarify correspondence between invariants and summaries
- tool construction guidelines
- examples with text-book verification challenges

Verification of `{ P } while t do B { Q }`

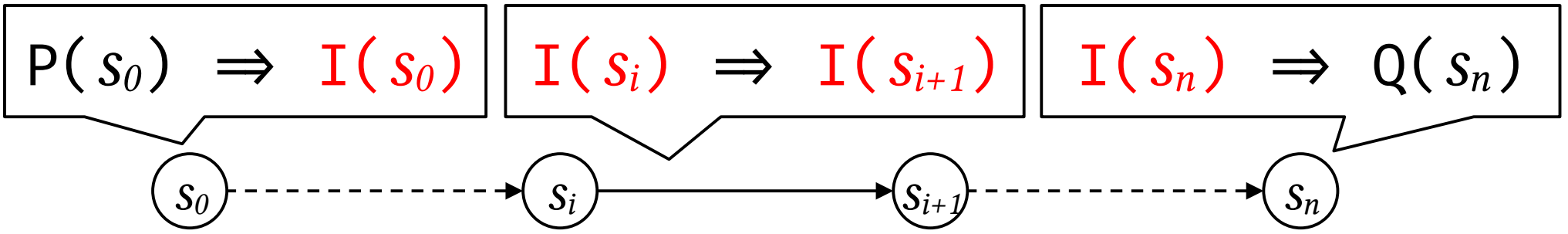


Verification of `{ P } while t do B { Q }`

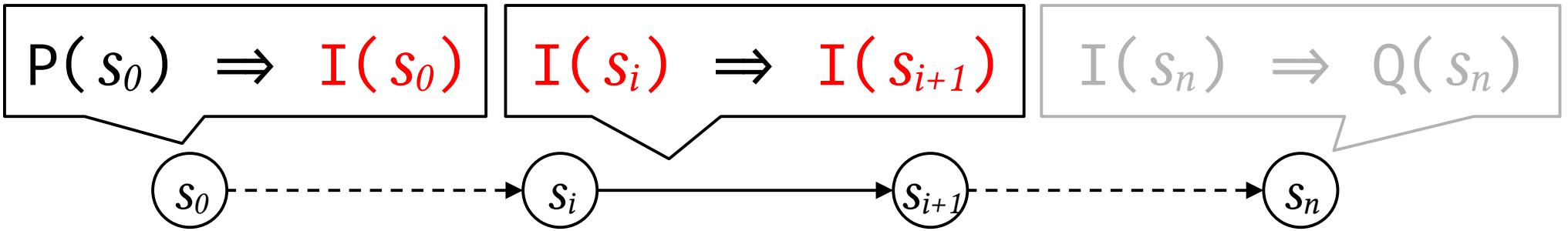


Tacitly assumed: $t(s_i)$ $\neg t(s_n)$

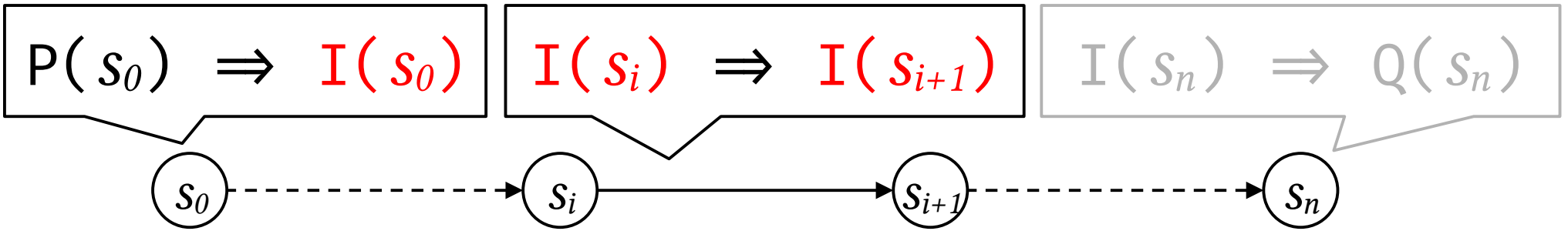
Verification of `{ P } while t do B { Q }`



Verification of `{ P } while t do B { Q }`

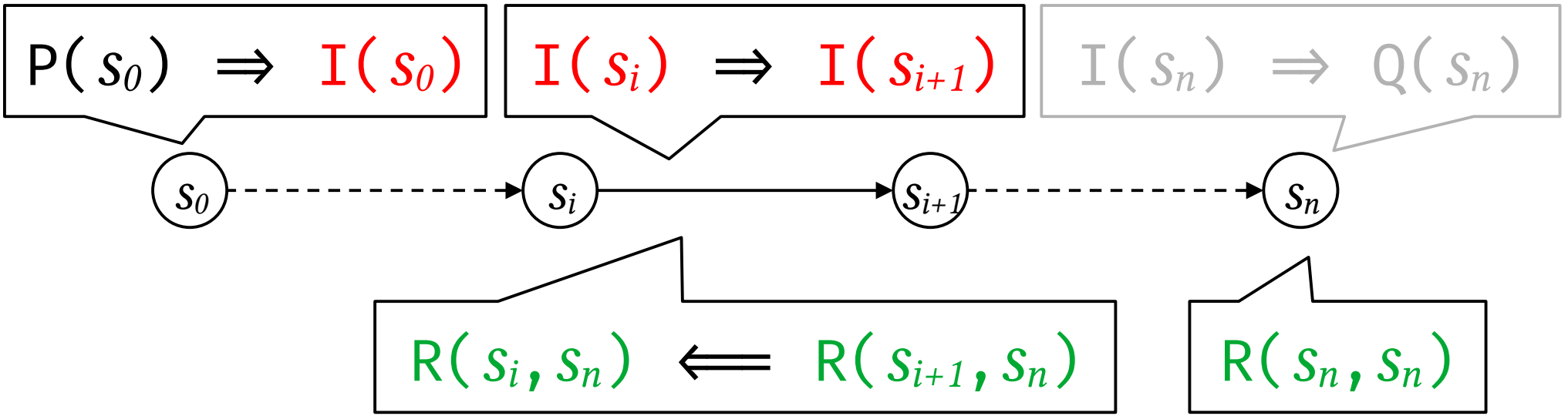


Verification of `{ P } while t do B { Q }`



$$P(s_0) \land R(s_0, s_n) \Longrightarrow Q(s_n)$$

Verification of `{ P } while t do B { Q }`



$$P(s_0) \land R(s_0, s_n) \Longrightarrow Q(s_n)$$

Loop Contracts [Hehner, Tuerk, …]

1. … generalize proofs with invariants
2. … sometimes capture proofs quite naturally

This paper: explain loop contracts

- *simple* formalization of verification conditions
- clarify correspondence between invariants and summaries
- syntactic proof rule for tool implementations
- examples with text-book verification challenges

Constructive Translations

1. “Canonical” summary recovers standard approach

$$R(s, s') := \neg t(s') \wedge I(s')$$

2. Encoding of summaries as part of invariants

$$I(s) := \exists\ s_0.\ P(s_0) \wedge (\forall\ s'.\ R(s, s') \implies R(s_0, s'))$$

Constructive Translations

1. “Canonical” summary recovers standard approach

$$R(s, s') := \neg t(s') \wedge I(s')$$

Result: characterization of completeness

summaries: functionality
invariants: no runtime errors / functionality
variants: termination

Constructive Translations

we can re-use existing tools with
"just" frontend engineering

(no excuse to not have loop contracts with summaries)

2. Encoding of summaries as part of invariants

$$I(s) := \exists\ s_0.\ P(s_0) \wedge (\forall\ s'.\ R(s, s') \implies R(s_0, s'))$$

Constructive Translations

Drawback: additional Quantifiers! But sometimes not needed:

∃ s_0 obsolete with `\old( ... )` in invariants

∀ s' can be eliminated for variables with `x' = f(s)`

2. Encoding of summaries as part of invariants

$$I(s) := \exists\ s_0.\ P(s_0) \wedge (\forall\ s'.\ R(s,s') \implies R(s_0,s'))$$

Loop Contract = Invariant + Summary

Invariant

- what has been done

Summary

- what is left to do

constructive translations:
can choose how to encode correctness properties

Key Insight:
can decouple *conceptual solution* from *technical approach*

Loop Contracts [Hehner, Tuerk, ...]

1. ... generalize proofs with invariants
2. ... sometimes capture proofs quite naturally

This paper: explain loop contracts

- *simple* formalization of verification conditions
- clarify correspondence between invariants and summaries
- syntactic proof rule for tool implementations
- examples with text-book verification challenges

Hoare Logic proof rule (same for `wp`/`sp`)

(treat loop as tail-recursive procedure)

$$\{\ I(x) \wedge x = x_0\ \}\ \texttt{while t do B}\ \{\ R(x_0, x)\ \}$$

Hoare Logic proof rule (same for `wp`/`sp`)

$$\frac{\{\ t(x) \wedge I(x) \wedge x = x_i\ \}\ B;\ x:[I, R]\ \{\ R(x_i, x)\ \} \qquad \{\ \neg\ t(x) \wedge I(x) \wedge x = x_n\ \}\ \texttt{skip}\ \{\ R(x_n, x)\ \}}{\{\ I(x) \wedge x = x_0\ \}\ \texttt{while t do B}\ \{\ R(x_0, x)\ \}}$$

Hoare Logic proof rule (same for `wp`/`sp`)

Key Idea:
specification statement [Morgan 88]
embeds application of inductive hypothesis into program

$$\frac{\{\ t(x) \wedge I(x) \wedge x = x_i\ \}\ B;\ \texttt{x: [I, R]}\ \{\ R(x_i, x)\ \} \qquad \{\ \neg\ t(x) \wedge I(x) \wedge x = x_n\ \}\ \texttt{skip}\ \{\ R(x_n, x)\ \}}{\{\ I(x) \wedge x = x_0\ \}\ \texttt{while t do B}\ \{\ R(x_0, x)\ \}}$$

Hoare Logic proof rule (same for wp/sp)

Key Idea:
specification statement [Morgan 88]
embeds application of inductive hypothesis into program

`assert I(x); `x_{i+1}`:= x; havoc x; assume R(`x_{i+1}`, x)`

$$\frac{\{\ t(x) \wedge I(x) \wedge x = x_i\ \}\ B;\ \texttt{x: [I, R]}\ \{\ R(x_i, x)\ \} \qquad \{\ \neg\ t(x) \wedge I(x) \wedge x = x_n\ \}\ \texttt{skip}\ \{\ R(x_n, x)\ \}}{\{\ I(x) \wedge x = x_0\ \}\ \texttt{while t do B}\ \{\ R(x_0, x)\ \}}$$

Loop Contracts [Hehner, Tuerk, ...]

1. ... generalize proofs with invariants
2. ... sometimes capture proofs quite naturally

This paper: explain loop contracts

- *simple* formalization of verification conditions
- clarify correspondence between invariants and summaries
- syntactic proof rule for tool implementations
- examples with text-book verification challenges (more: paper)

Example: `bsearch(int x, int[] a)`

invariant $x \notin a[0..l] \land x \notin a[r..a.length]$

summary $\result \iff x \in a[l..r]$

Example: `bsearch(int x, int[] a)`

invariant `x ∉ a[0..l] ∧ x ∉ a[r..a.length]`

summary `\result ⟺ x ∈ a[l..r]`

Comparison: summary often captures *intention* really well

Example: `bsearch(int x, int[] a)`

invariant $x \notin a[0..l] \land x \notin a[r..a.length]$

summary $\backslash result \iff x \in a[l..r]$

Comparison: summary often captures *intention* really well

summary encoded as invariant and simplified

$$x \in a[l..r] \iff a[0..a.length]$$

Example: “right-fold” specifications

$$f :: State \longrightarrow List$$

$$f(s) = \begin{cases} [] & \text{if } \neg\, t(s) \\ x :: f(s') & \text{if } \quad t(s) \wedge B(s,s') \end{cases}$$

typical specification pattern:
abstraction with algebraic lists produces
first element in recursion (“right-fold”)

Example: "right-fold" specifications

```
f :: State ⟶ List
```

$$f(s) = \begin{cases} [] & \text{if } \neg\, t(s) \\ x :: f(s') & \text{if } \quad t(s) \wedge B(s,s') \end{cases}$$

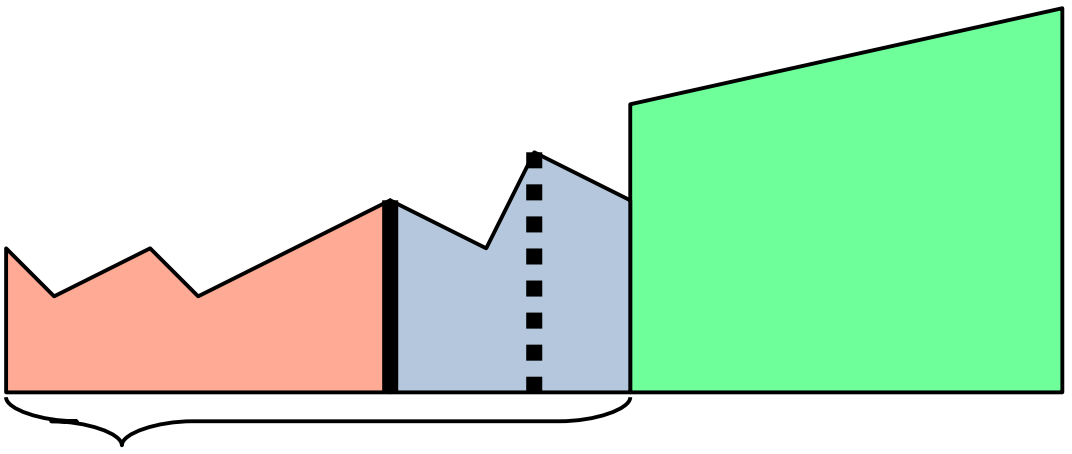
Comparison:

Invariants: need lemma to translate `f` into *left-fold*

Summaries: matches *right-fold* `f` **directly**

(cf. phone number example in the paper)

Example: Bubble Sort

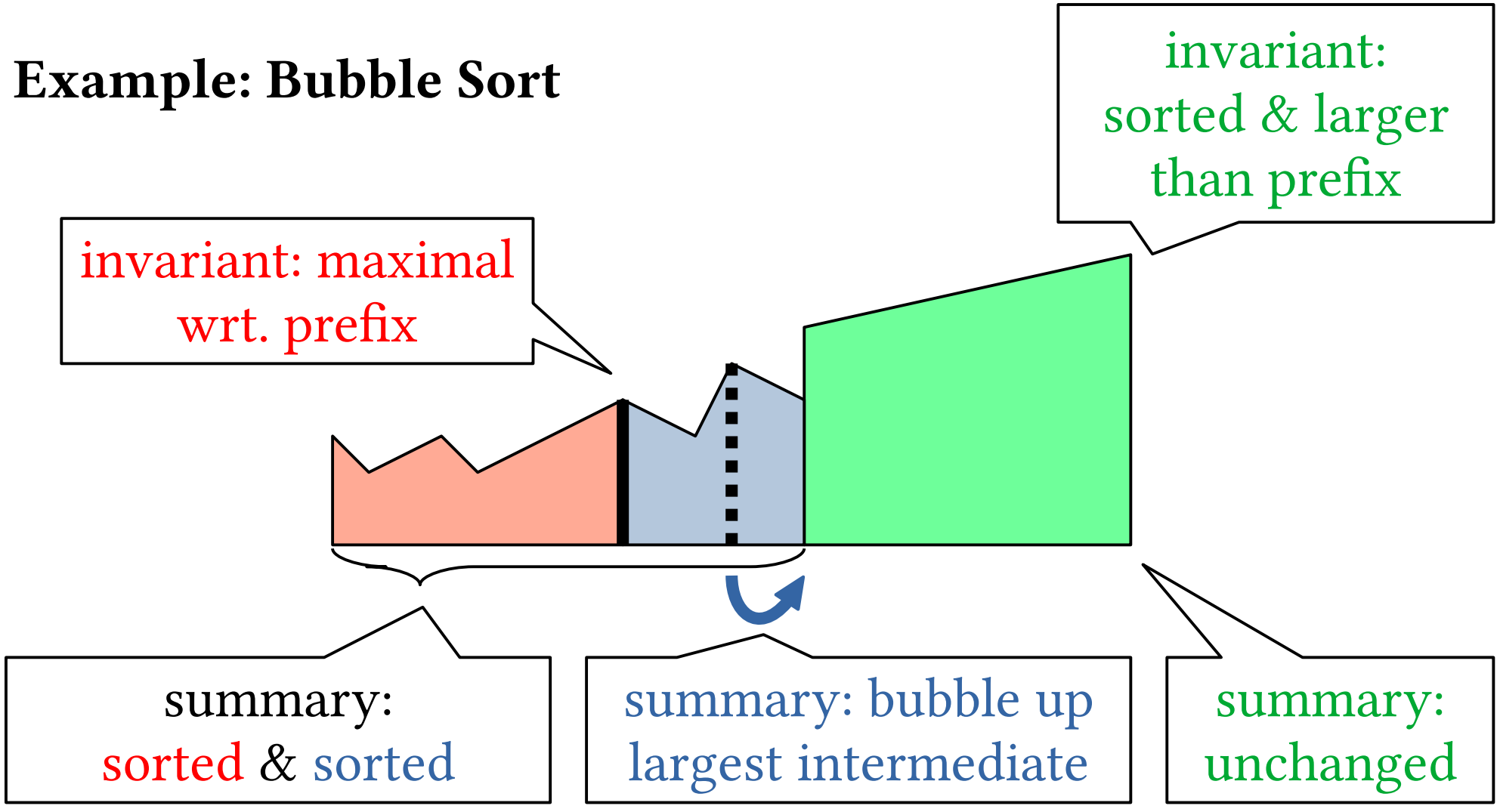


invariant:
sorted & larger
than prefix

summary:
sorted & sorted

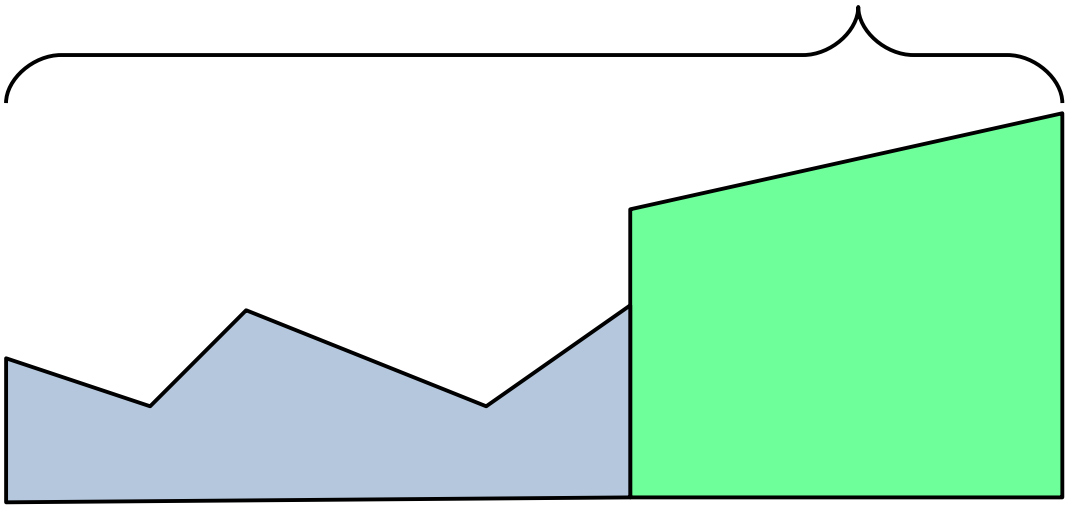
summary:
unchanged

Example: Bubble Sort



Example: Insertion Sort

summary: overall sorted
(= canonical summary)



invariant and also precondition for contract:
sorted & larger than prefix

Comparison of Sorting Specifications

Bubble Sort

- invariant and summary nicely symmetrical
- work to be done *independent* from work done so far

Insertion Sort

- work to be done depends on sorted suffix
- both approaches essentially the same

Preliminary: Automatic Inference of Summaries?

- Generalize postconditions by initial values:

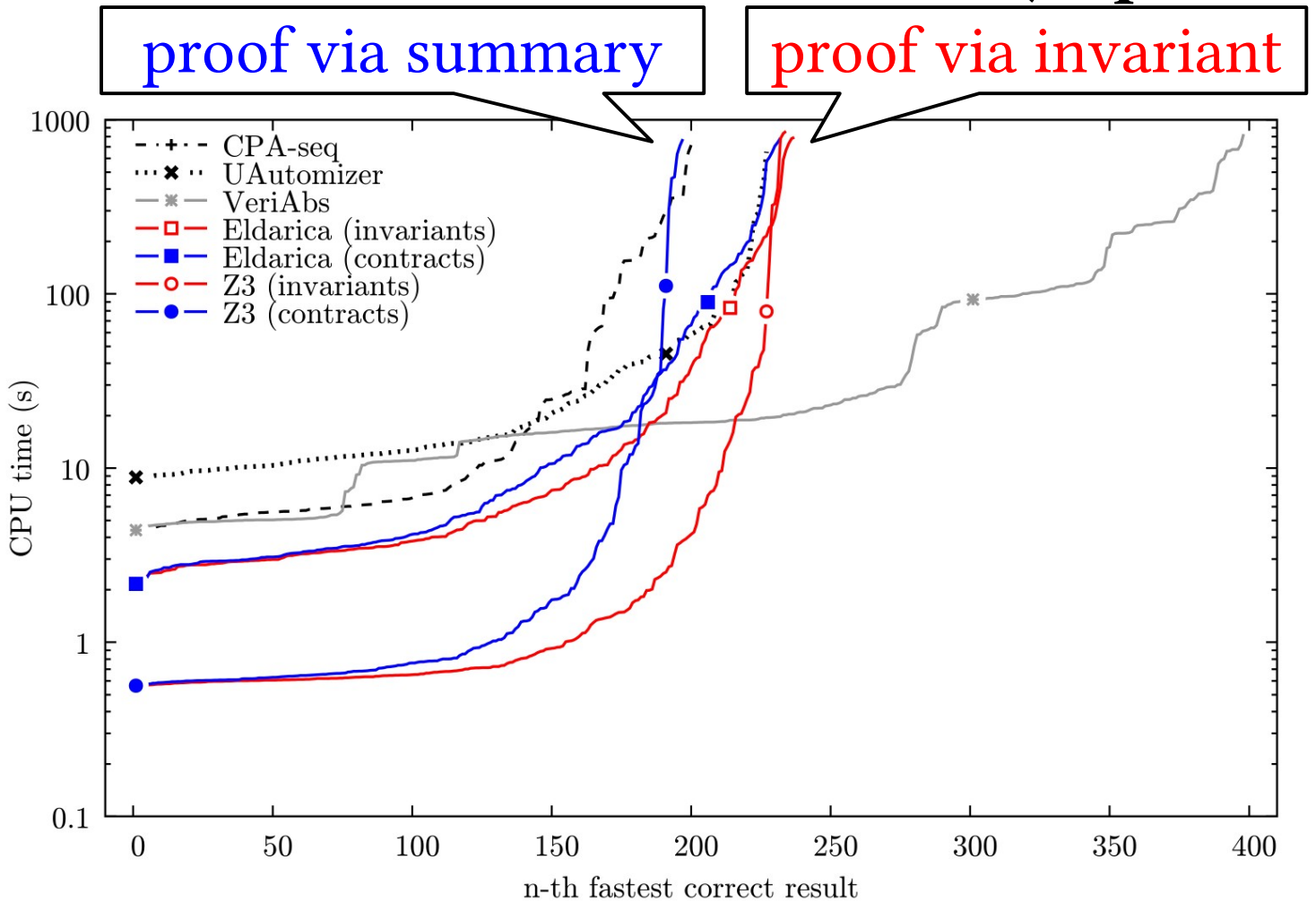
  `max(a[0..a.length])` ⇝ `max(a[l..r])`

  → works for simple examples only

- Experiment: ask Horn solvers

  Inference of unknown predicates: Eldarica, Z3, …

Off-the-shelf Horn Solvers (Experiments from 2020)



Evaluation like SV-Comp 2020

Categories: Loops, Recursive, Arrays, ControlFlow

Preliminary: Automatic Inference of Summaries?

- Generalize postconditions by initial values:

  `max(a[0..a.length])` ⇝ `max(a[l..r])`

  → works for simple examples only

- Experiment: ask Horn solvers

  solver improvements + probably bad encoding
  → need to re-run the experiments!

Example: `copy(int[] a, int[] b)`

invariant `b[..i] = a[..i]`

summary `final(b)[i..] = a[i..]` $\wedge$ ...

Example: `copy(int[] a, int[] b)`

invariant `b[..i] = a[..i]`

summary `final(b)[i..] = a[i..]` ∧

`final(b)[..i] = b[..i]`

Comparison: summary may require *additional framing*

completing the loop "forgets" leading `b[i] := a[i]`

Frame Conditions (Preliminary Ideas)

```
while i < n do
   a[i] := z; i := i+1
```

cf. Transition Invariants
[Podelstki, Rybalchenko 04]

```
final(a[k]) == a[k]
    if ???
```

Frame Conditions (Preliminary Ideas)

```
while i < n do
   a[i] := z; i := i+1
```

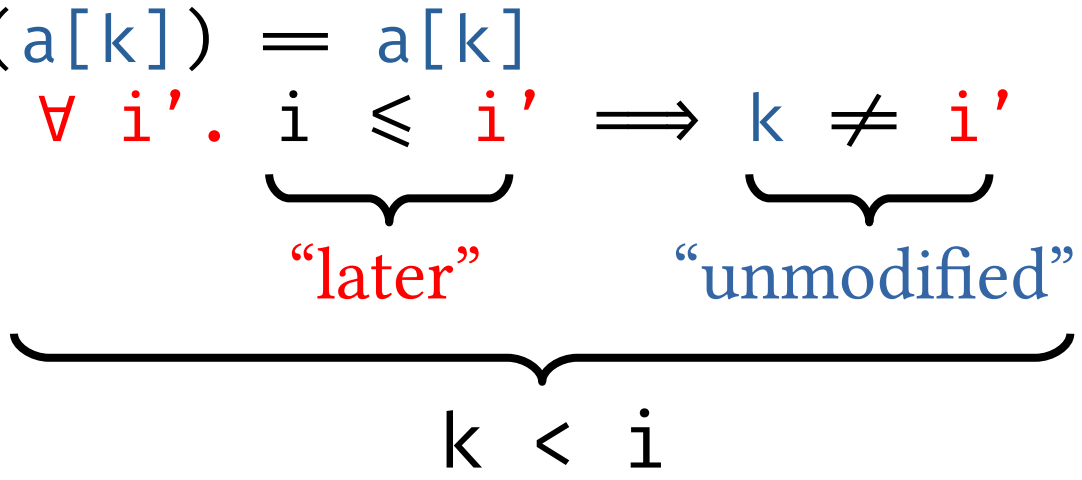
cf. Transition Invariants
[Podelstki, Rybalchenko 04]

$$\texttt{final(a[k])} = \texttt{a[k]} \quad \text{if } \forall\, \texttt{i'}.\ \underbrace{\texttt{i} \leqslant \texttt{i'}}_{\text{“later”}} \implies \underbrace{\texttt{k} \neq \texttt{i'}}_{\text{“unmodified”}}$$

Frame Conditions (Preliminary Ideas)

```
while i < n do
   a[i] := z; i := i+1
```

cf. Transition Invariants
[Podelstki, Rybalchenko 04]



Demo: Cuvée

https://github.com/gernst/cuvee

Thanks for your attention!

Loop Contract

Invariant	Summary
well-known & researched	→ full potential to be explored
forwards / left-fold	backwards / right-fold
fewer frame conditions	similar to postcondition (= intention)

Conclusion: Want both! Theory & Tool support is easy.

Summaries propagate backwards

$$R(s_i, s_n) \Longleftarrow R(s_{i+1}, s_n)$$

- case `if a[l] ⩽ a[r] then l := l+1`

Summaries propagate backwards

$$R(s_i, s_n) \Longleftarrow R(s_{i+1}, s_n)$$

- case `if a[l] ⩽ a[r] then l := l+1`
  - assume `a[final(l)] = max(a[l+1..r+1])`

Summaries propagate backwards

$$R(s_i, s_n) \Longleftarrow R(s_{i+1}, s_n)$$

- case `if a[l] ⩽ a[r] then l := l+1`
  - assume `a[final(l)] = max(a[l+1 .. r+1])`
  - therefore `a[final(l)] ⩾ a[r]`

`a[r]` is part of `a[l+1 .. r+1]`

Summaries propagate backwards

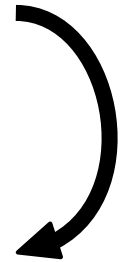
$$R(s_i, s_n) \Longleftarrow R(s_{i+1}, s_n)$$

- case `if a[l] ⩽ a[r] then l := l+1`
  - assume `a[final(l)] = max(a[l+1..r+1])`
  - therefore `a[final(l)] ⩾ a[r]`
  - prove `a[final(l)] = max(a[l..r+1])`

Summaries propagate backwards

$$R(s_i, s_n) \Longleftarrow R(s_{i+1}, s_n)$$

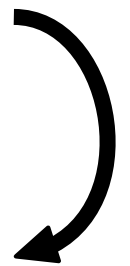
- case `if a[l] ⩽ a[r] then l := l+1`
  - assume `a[final(l)] = max(a[l+1 .. r+1])`
  - therefore `a[final(l)] ⩾ a[r]`
  - prove `a[final(l)] = max(a[l .. r+1])`



Summaries propagate backwards

$$R(s_i, s_n) \Longleftarrow R(s_{i+1}, s_n)$$

- case `if a[l] ⩽ a[r] then l := l+1`
  - assume `a[final(l)] = max(a[l+1..r+1])`
  - therefore `a[final(l)] ⩾ a[r]`
  - prove `a[final(l)] = max(a[l..r+1])`
  - because `a[final(l)] ⩾ ... ⩾ a[l]`



Summaries propagate backwards

$$R(s_i, s_n) \Longleftarrow R(s_{i+1}, s_n)$$

- case `if a[l] ⩽ a[r] then l := l+1`
  - assume `a[final(l)] = max(a[l+1..r+1])`
  - therefore `a[final(l)] ⩾ a[r]`
  - prove `a[final(l)] = max(a[l..r+1])`
  - because `a[final(l)] ⩾ a[r] ∧ a[r] ⩾ a[l]`