Separating Functional and Parallel Correctness using Nondeterministic Sequential Specifications

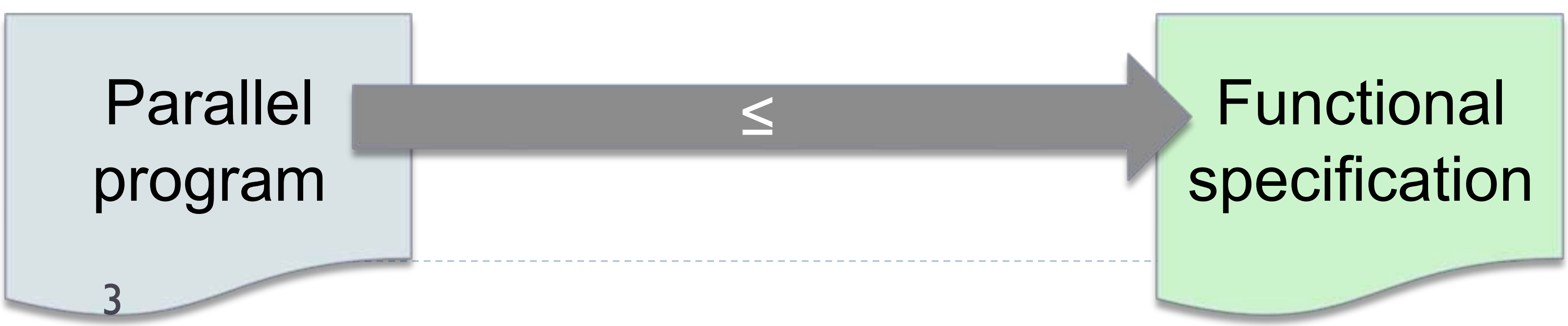
Jacob Burnim, George Necula, Koushik Sen
University of California, Berkeley

Parallel Programming is Hard

- **Key Culprit:** Nondeterministic interleaving of parallel threads.
  - Painful to reason simultaneously about parallelism and functional correctness.
- **Goal:** Decompose efforts in addressing parallelism and functional correctness.
  - Allow programmers to reason about functional correctness **sequentially**.
  - Independently show correctness of parallelism.

Our Approach

- **Goal:** Decompose efforts in addressing parallelism and functional correctness.

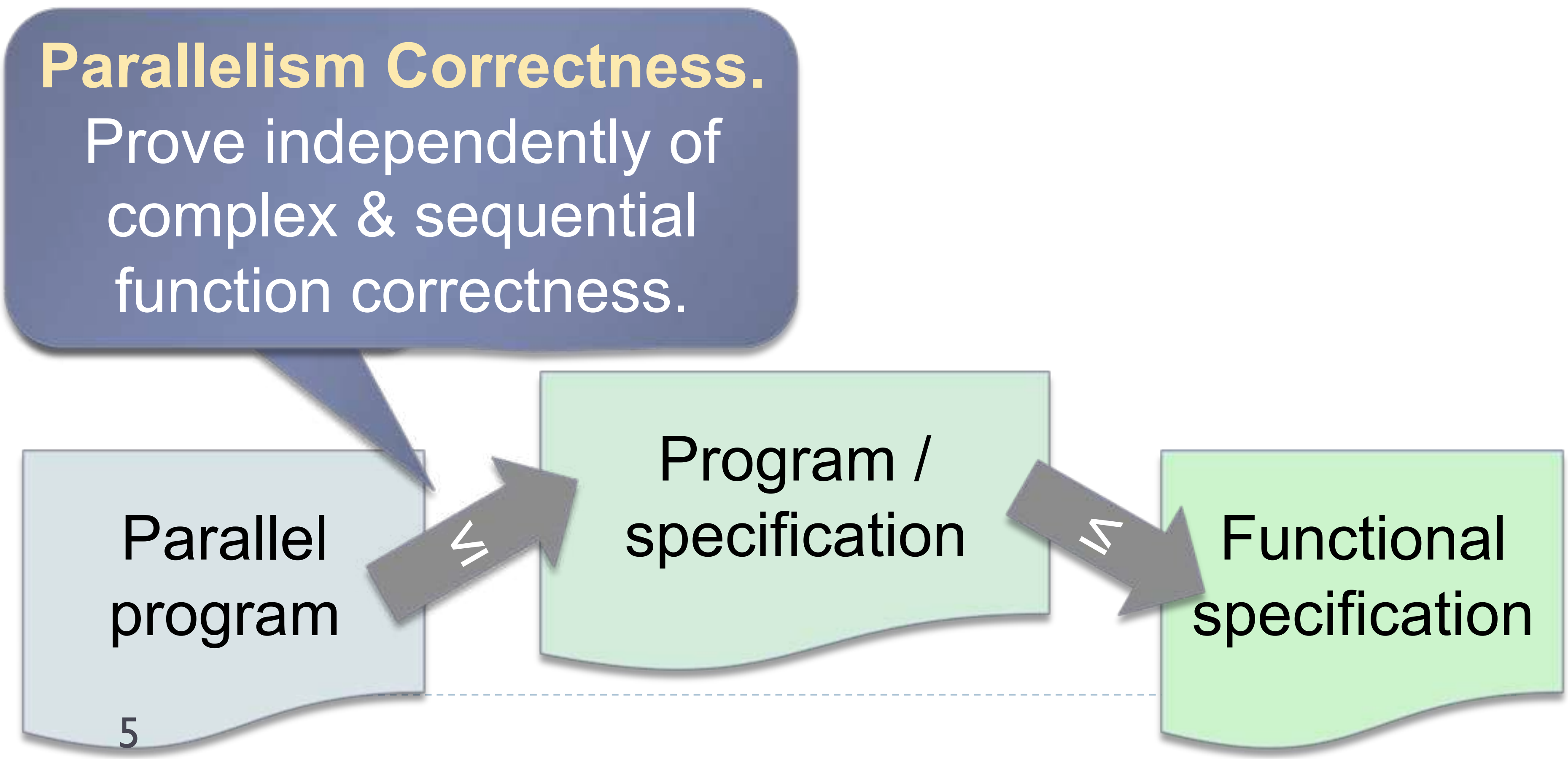


Our Approach

- **Goal:** Decompose efforts in addressing parallelism and functional correctness.

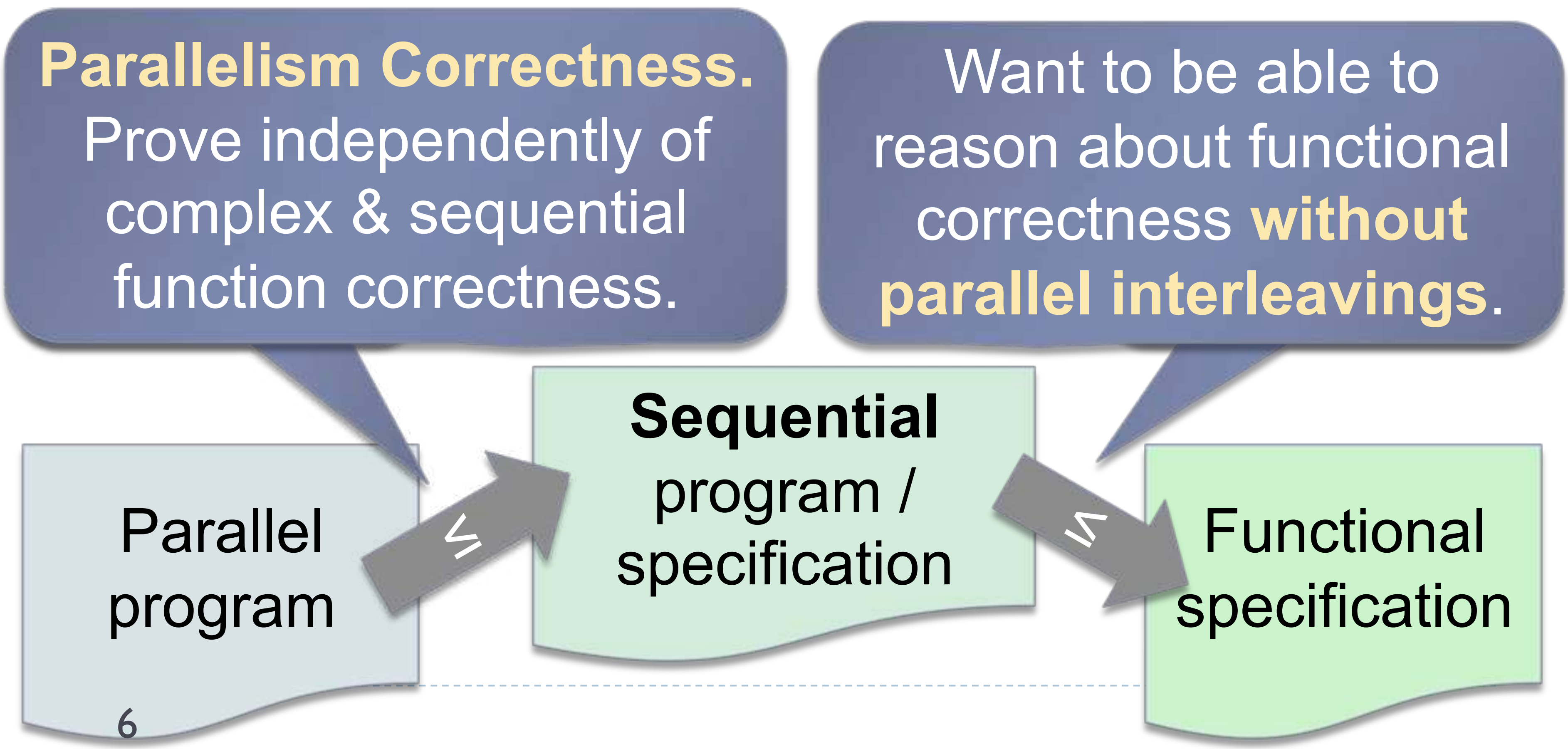
Our Approach

- **Goal:** Decompose efforts in addressing parallelism and functional correctness.



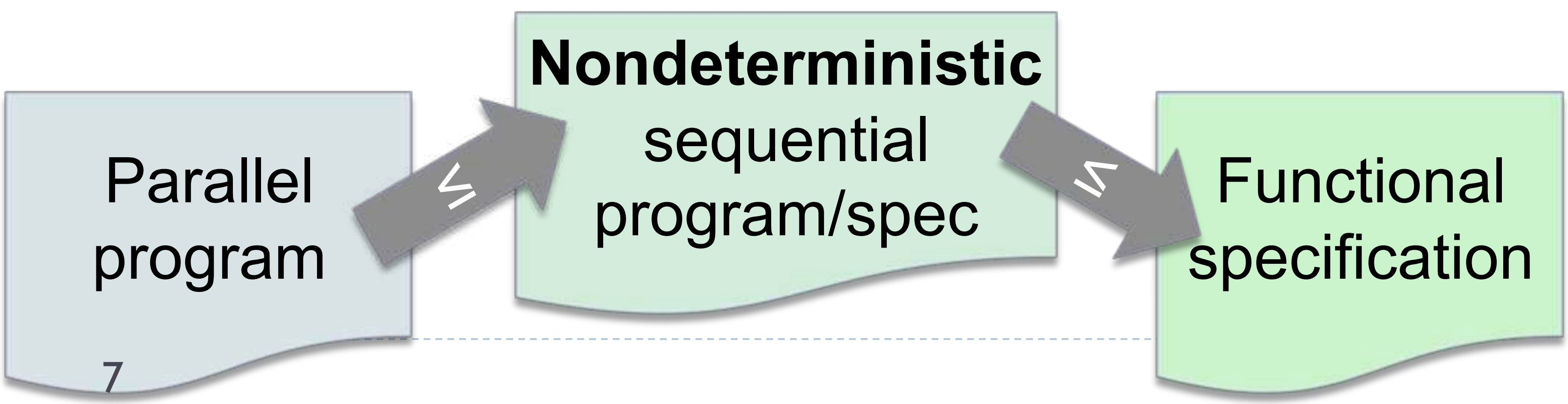
Our Approach

- **Goal:** Decompose efforts in addressing parallelism and functional correctness.



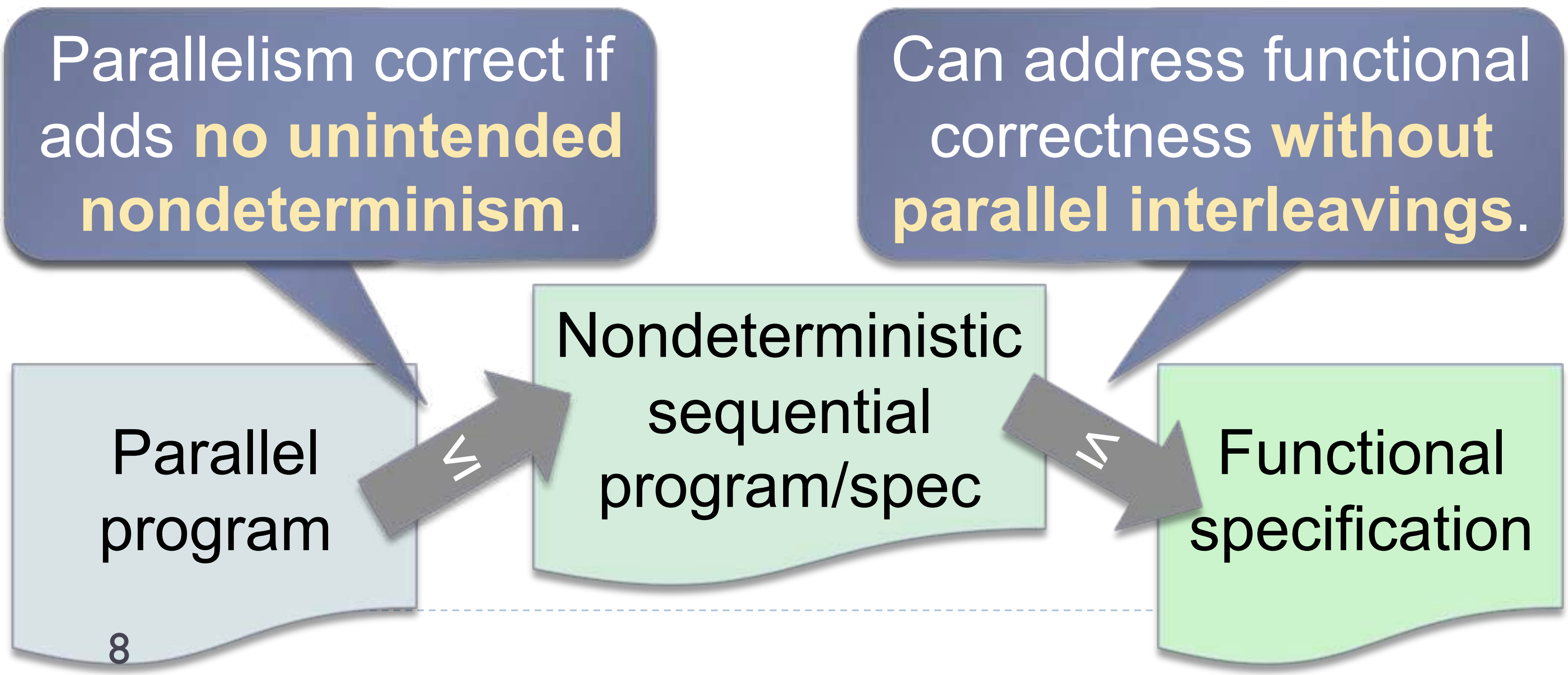
Our Approach

- Use **sequential but nondeterministic** specification for a program's parallelism.
  - User annotates **intended nondeterminism**.



Our Approach

- Use **sequential but nondeterministic** specification for a program's parallelism.
  - User annotates **intended nondeterminism**.



Outline

- Overview
- **Motivating Example**
- Nondeterministic Sequential (NDSEQ) Specifications for Parallel Correctness
- Proving Parallel Correctness
- Future Work
- Conclusions

Motivating Example

- **Goal:** Find minimum-cost solution.
  - Simplified branch-and-bound benchmark.

```
for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  if cost < best:
    best = cost
    best_soln = w
```

Motivating Example

- **Goal:** Find minimum-cost solution.
  - Simplified branch-and-bound benchmark.

```
for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  if cost < best:
    best = cost
    best_soln = w
```

Input: List of possible solutions.

Output: Solution from input queue with minimum cost.

Motivating Example

- **Goal:** Find minimum-cost solution.
  - Simplified branch-and-bound benchmark.

```
for (w in queue):
  if (lower_bnd(w) >= best):
     continue
  cost = compute_cost(w)
  if cost < best:
     best = cost
     best_soln = w
```

Computes cost of solution w. **Expensive**.

Motivating Example

- **Goal:** Find minimum-cost solution.
  - Simplified branch-and-bound benchmark.

```
for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  if cost < best:
    best = cost
    best_soln = w
```

Computes **cheap** lower bound on cost of w.

Computes cost of solution w. **Expensive**.

Motivating Example

- **Goal:** Find minimum-cost solution.
  - Simplified branch-and-bound benchmark.

```
for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  if cost < best:
    best = cost
    best_soln = w
```

Computes **cheap** lower bound on cost of w.

Prune when w cannot have minimum-cost.

Computes cost of solution w. **Expensive**.

Motivating Example

queue:

(a)	bound: 1 cost: 2

(b)	bound: 0 cost: 3

(c)	bound: 5 cost: 9

best: ∞

best_soln: •

```
for (w in queue):
  if (lower_bnd(w) >= best):
     continue
  cost = compute_cost(w)
  if cost < best:
     best = cost
     best_soln = w
```

Motivating Example

queue:

(a) bound: 1 cost: 2	(b) bound: 0 cost: 3	(c) bound: 5 cost: 9

best: ∞

best_soln: •

```
for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  if cost < best:
    best = cost
    best_soln = w
```

prune?(a)

Motivating Example

queue: (a) bound: 1 cost: 2 | (b) bound: 0 cost: 3 | (c) bound: 5 cost: 9

best: 2

best_soln: → (a)

```
for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  if cost < best:
    best = cost
    best_soln = w
```

prune?(a)

update(a)

Motivating Example

queue: (a) bound: 1 cost: 2 | (b) bound: 0 cost: 3 | (c) bound: 5 cost: 9

best: 2

best_soln: (a)

```
for (w in queue):
    if (lower_bnd(w) >= best):
        continue
    cost = compute_cost(w)
    if cost < best:
        best = cost
        best_soln = w
```

prune?(**a**)

update(**a**)

prune?(**b**)

Motivating Example

queue: **(a)** bound: 1 cost: 2 | **(b)** bound: 0 cost: 3 | **(c)** bound: 5 cost: 9

best: 2

best_soln: → (a)

```
for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  if cost < best:
    best = cost
    best_soln = w
```

prune?(**a**)

update(**a**)

prune?(**b**)

update(**b**)

Motivating Example

queue: (**a**) bound: 1 cost: 2 | (**b**) bound: 0 cost: 3 | (**c**) bound: 5 cost: 9

best: 2

best_soln: → (a)

```
for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  if cost < best:
    best = cost
    best_soln = w
```

prune?(**a**)

update(**a**)

prune?(**b**)

update(**b**)

prune?(**c**)

Motivating Example

queue: (a) bound: 1 cost: 2 | (b) bound: 0 cost: 3 | (c) bound: 5 cost: 9

best: 2

best_soln: → (a)

```
for (w in queue):
  if (lower_bnd(w) >= best):
     continue
  cost = compute_cost(w)
  if cost < best:
     best = cost
     best_soln = w
```

prune?(**a**)

update(**a**)

prune?(**b**)

update(**b**)

prune?(**c**)

Motivating Example

- **Goal:** Find minimum-cost solution.
  - Simplified branch-and-bound benchmark.

```
for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  if cost < best:
    best = cost
    best_soln = w
```

Motivating Example

How do we parallelize this code?

```
for (w in queue):
  if (lower_bnd(w) >= best):
     continue
  cost = compute_cost(w)
  if cost < best:
     best = cost
     best_soln = w
```

Parallelizing our Example

- **Goal:** Find min-cost solution **in parallel**.
  - Simplified branch-and-bound benchmark.

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

Parallelizing our Example

- **Goal:** Find min-cost solution **in parallel**.
  - Simplified branch-an

Loop iterations can be run in parallel.

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
     continue
  cost = compute_cost(w)
  atomic:
     if cost < best:
        best = cost
        best_soln = w
```

Updates to best are **atomic.**

Prove Parallelism Correct?

- **Claim:** Parallelization is correct.
  - If there are any bugs, they are **sequential**.
  - Want to prove **parallelization** correct.

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

Prove Parallelism Correct?

- **Claim:** Parallelization is correct.
  - If there are any bugs, they are **sequential**.

Idea: Specify that parallel version gives same result as sequential.

```
cost = compute_cost(w)
atomic:
    if cost < best:
        best = cost
        best_soln = w
```

Parallel-Sequential Equivalence?

queue:

(a)	(b)	(c)
bound: 1 cost: 2	bound: 0 cost: 2	bound: 5 cost: 9

best: ∞

best_soln: •

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

Parallel-Sequential Equivalence?

queue: (a) bound: 1 cost: 2 | (b) bound: 0 cost: 2 | (c) bound: 5 cost: 9

best: 2

best_soln: → (a)

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

prune?(**a**)

update(**a**)

Parallel-Sequential Equivalence?

queue: (**a**) bound: 1 cost: 2 | (**b**) bound: 0 cost: 2 | (**c**) bound: 5 cost: 9

best: 2

best_soln: (**a**)

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)
  atomic:
      if cost < best:
          best = cost
          best_soln = w
```

prune?(**a**)

update(**a**)

prune?(**b**)

update(**b**)

Parallel-Sequential Equivalence?

queue: (a) bound: 1 cost: 2 | (b) bound: 0 cost: 2 | (c) bound: 5 cost: 9

best: 2

best_soln: → (a)

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
     continue
  cost = compute_cost(w)
  atomic:
     if cost < best:
        best = cost
        best_soln = w
```

prune?(**a**)

update(**a**)

prune?(**b**)

update(**b**)

prune?(**c**)

Parallel-Sequential Equivalence?

queue: (**a**) bound: 1 cost: 2 | (**b**) bound: 0 cost: 2 | (**c**) bound: 5 cost: 9

best: 2

best_soln: •

Sequential program always finds **best_soln = (a)**.

prune?(**a**)

update(**a**)

prune?(**b**)

update(**b**)

prune?(**c**)

Parallel-Sequential Equivalence?

queue:

(a) bound: 1 cost: 2	(b) bound: 0 cost: 2	(c) bound: 5 cost: 9

best: ∞

best_soln: •

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
     continue
  cost = compute_cost(w)
  atomic:
     if cost < best:
        best = cost
        best_soln = w
```

Parallel-Sequential Equivalence?

queue: (a) bound: 1 cost: 2 | (b) bound: 0 cost: 2 | (c) bound: 5 cost: 9

best: ∞

best_soln: •

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

prune?(a)

Parallel-Sequential Equivalence?

queue: **(a)** bound: 1 cost: 2 | **(b)** bound: 0 cost: 2 | **(c)** bound: 5 cost: 9

best: 2

best_soln: → (b)

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

prune?(**a**)

prune?(**b**)

update(**b**)

Parallel-Sequential Equivalence?

queue: (**a**) bound: 1 cost: 2 | (**b**) bound: 0 cost: 2 | (**c**) bound: 5 cost: 9

best: 2

best_soln: → (**b**)

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

prune?(**a**)

prune?(**b**)

update(**b**)

prune?(**c**)

Parallel-Sequential Equivalence?

queue: (**a**) bound: 1 cost: 2 | (**b**) bound: 0 cost: 2 | (**c**) bound: 5 cost: 9

best: 2

best_soln: (**b**)

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

- prune?(**a**)
- prune?(**b**)
- update(**b**)
- prune?(**c**)
- update(**a**)

Parallel-Sequential Equivalence?

queue: (**a**) bound: 1 cost: 2 | (**b**) bound: 0 cost: 2 | (**c**) bound: 5 cost: 9

best: 2

best_soln: (**b**)

Parallel version can also find **best_soln = (b).**

prune?(**a**)

prune?(**b**)

update(**b**)

prune?(**c**)

update(**a**)

Parallel-Sequential Equivalence?

- Parallel and sequential **not** equivalent.
  - **Claim:** But parallelism is correct.

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)
  atomic:
      if cost < best:
          best = cost
          best_soln = w
```

Parallel-Sequential Equivalence?

- Parallel and sequential **not** equivalent.
  - **Claim:** But parallelism is correct.

Some nondeterminism is okay.

Specification for the **parallelism** must indicate **intended** or **algorithmic** nondeterminism.

```
p
  ...
  if cost < best:
      best = cost
      best_soln = w
```

NDSEQ Specification

- Use nondeterministic sequential (**NDSEQ**) version of program as spec for parallelism.

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)
  atomic:
      if cost < best:
          best = cost
          best_soln = w
```

```
nondet-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)

  if cost < best:
      best = cost
      best_soln = w
```

NDSEQ Specification

- Use **NDSEQ**)
  vers rallelism.

Allow sequential code to perform iterations in a nondeterministic order.

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)
  atomic:
      if cost < best:
          best = cost
          best_soln = w
```

```
nondet-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)

  if cost < best:
      best = cost
      best_soln = w
```

NDSEQ Specification

- Specifies:
  - For every parallel execution, there must exist an NDSEQ execution with the same result.

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)
  atomic:
      if cost < best:
          best = cost
          best_soln = w
```

```
nondet-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)

  if cost < best:
      best = cost
      best_soln = w
```

Parallel-NDSEQ Equivalence?

queue: (**a**) bound: 1 cost: 2 | (**b**) bound: 0 cost: 2 | (**c**) bound: 5 cost: 9

best: 2

best_soln: → (**b**)

Parallel:

prune?(**a**)

prune?(**b**)

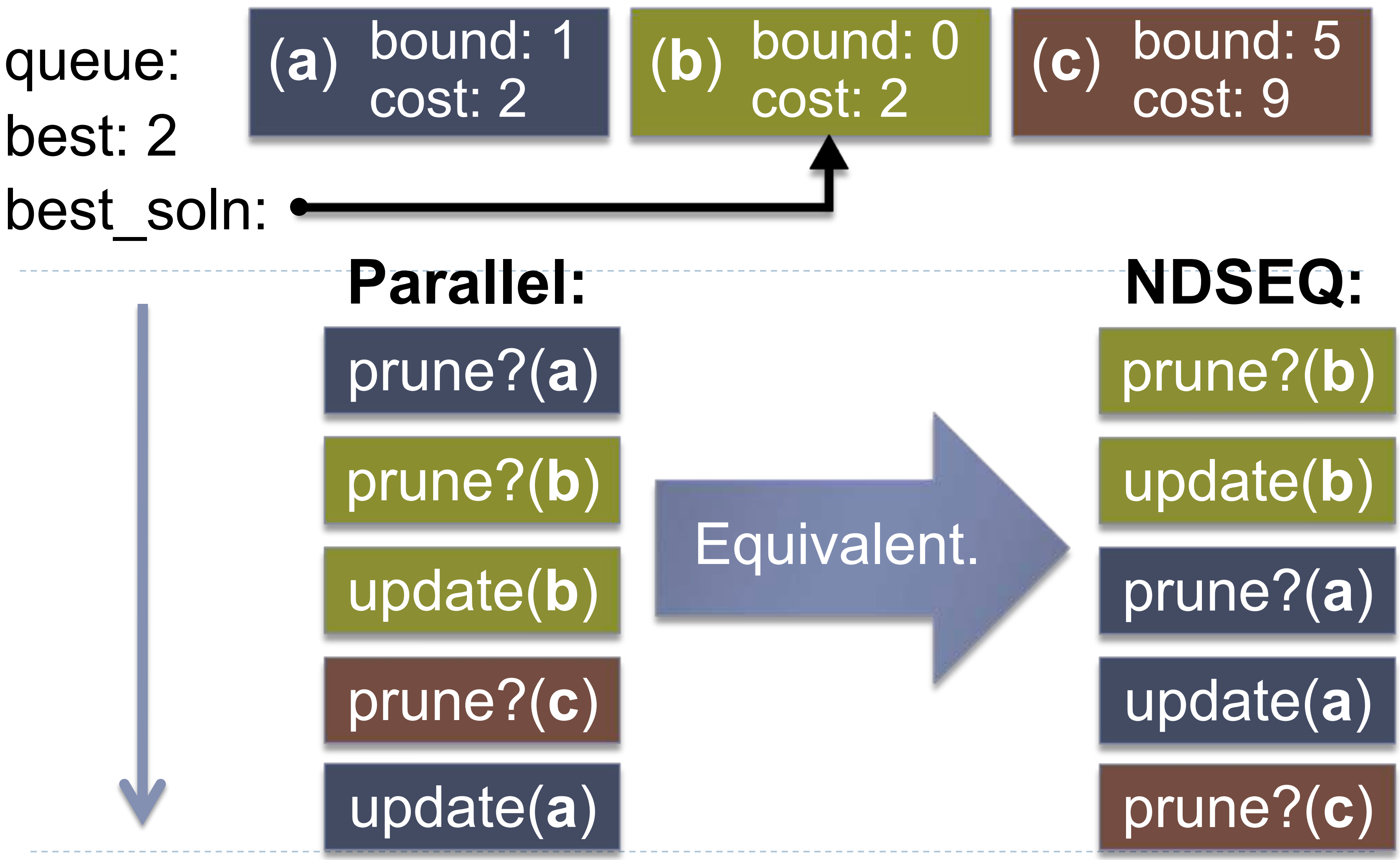
update(**b**)

prune?(**c**)

update(**a**)

- No equivalent sequential execution.
- An equivalent NDSEQ execution?

Parallel-NDSEQ Equivalence?



NDSEQ Specification

Does this NDSEQ specification really capture correctness of the parallelism?

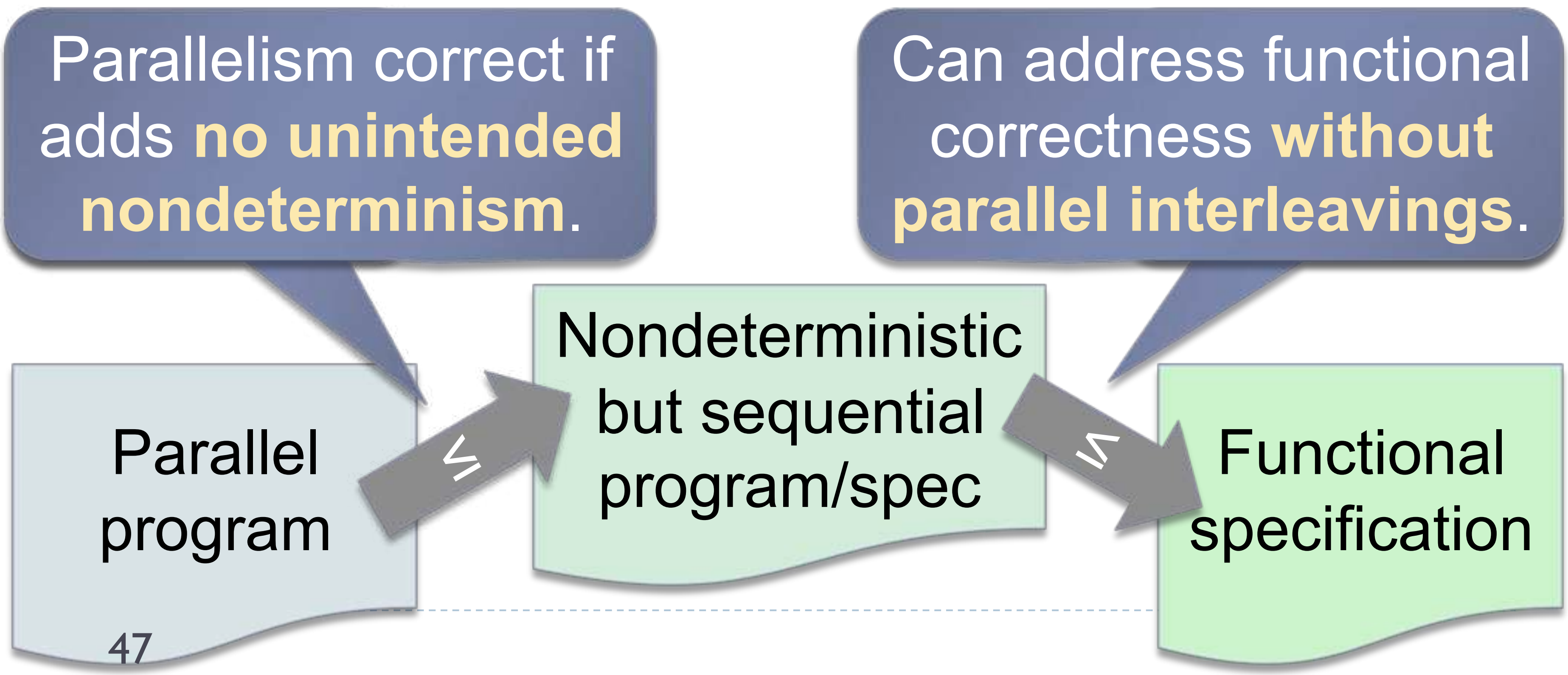
```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)
  atomic:
      if cost < best:
          best = cost
          best_soln = w
```

```
nondet-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)

  if cost < best:
      best = cost
      best_soln = w
```

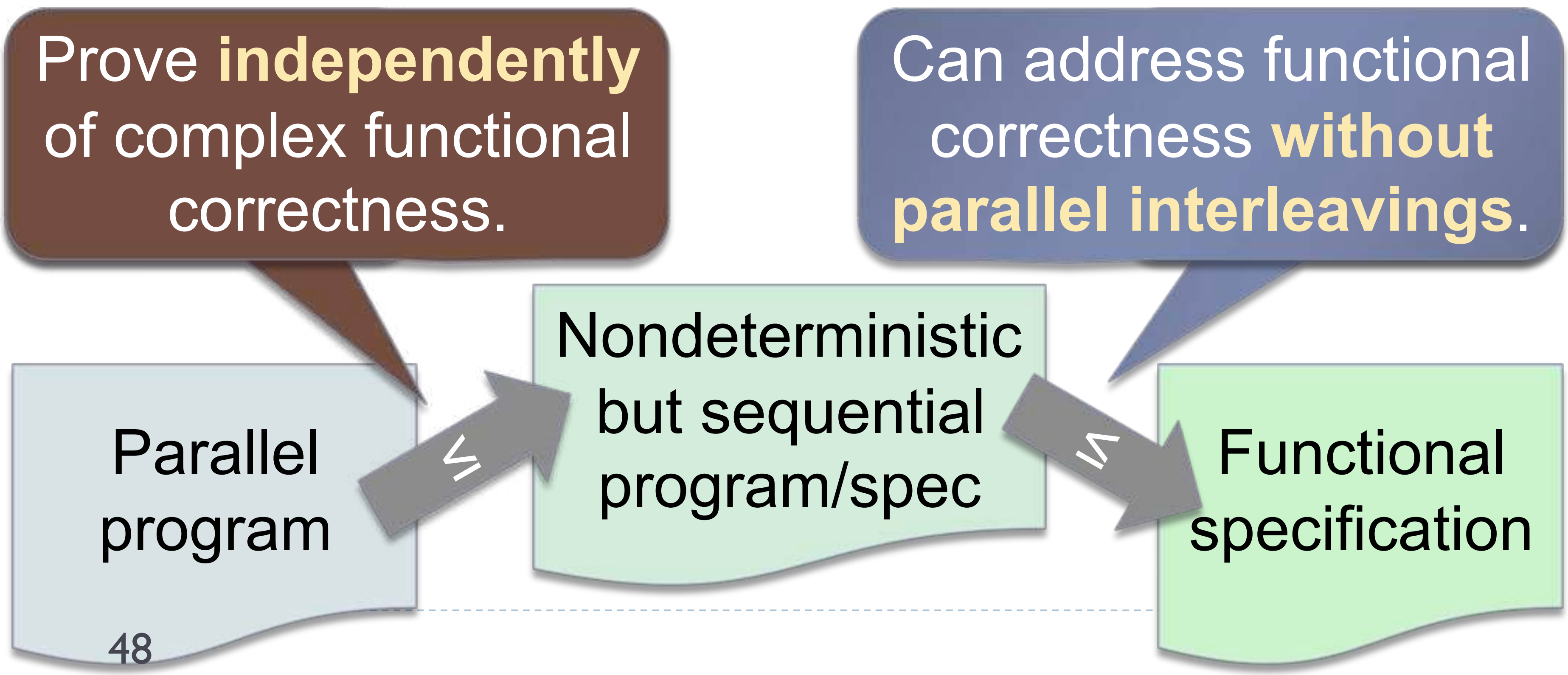
Recall: Our Approach

- Use **sequential but nondeterministic** specification for a program's parallelism.
  - User annotates **intended nondeterminism**.



Recall: Our Approach

- Use **sequential but nondeterministic** specification for a program's parallelism.
  - User annotates **intended nondeterminism**.



Parallel-NDSEQ Equivalence?

queue: **(a)** bound: 2 cost: 2 | **(b)** bound: 2 cost: 2 | **(c)** bound: 5 cost: 9

best: ∞

best_soln: •

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
     continue
  cost = compute_cost(w)
  atomic:
     if cost < best:
        best = cost
        best_soln = w
```

Parallel-NDSEQ Equivalence?

queue: (a) bound: 2 cost: 2 | (b) bound: 2 cost: 2 | (c) bound: 5 cost: 9

best: ∞

best_soln: •

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

prune?(a)

prune?(b)

Parallel-NDSEQ Equivalence?

queue: (a) bound: 2 cost: 2 | (b) bound: 2 cost: 2 | (c) bound: 5 cost: 9

best: 2

best_soln: → (a)

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
     continue
  cost = compute_cost(w)
  atomic:
     if cost < best:
        best = cost
        best_soln = w
```

prune?(**a**)

prune?(**b**)

update(**a**)

Parallel-NDSEQ Equivalence?

queue:

(a) bound: 2 cost: 2	(b) bound: 2 cost: 2	(c) bound: 5 cost: 9

best: 2

best_soln: → (a)

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

prune?(a)

prune?(b)

update(a)

update(b)

Parallel-NDSEQ Equivalence?

queue: **(a)** bound: 2, cost: 2 | **(b)** bound: 2, cost: 2 | **(c)** bound: 5, cost: 9

best: 2

best_soln: → **(a)**

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)
  atomic:
      if cost < best:
          best = cost
          best_soln = w
```

prune?(**a**)

prune?(**b**)

update(**a**)

update(**b**)

prune?(**c**)

Parallel-NDSEQ Equivalence?

queue:
best: 2
best_soln:

(a) bound: 2 cost: 2

(b) bound: 2 cost: 2

(c) bound: 5 cost: 9

Parallel code can avoid pruning by interleaving iterations.

NDSEQ version must prune either (**a**) or (**b**).

prune?(**a**)

prune?(**b**)

update(**a**)

update(**b**)

prune?(**c**)

Parallel-NDSEQ Equivalence?

queue: (**a**) bound: 2 cost: 2 | (**b**) bound: 2 cost: 2 | (**c**) bound: 5 cost: 9

best: 2

best_soln: → (**a**)

Parallel code can avoid pruning by interleaving iterations.

NDSEQ **should** have freedom to **not** prune.

prune?(**a**)

prune?(**b**)

update(**a**)

update(**b**)

prune?(**c**)

NDSEQ Specification

Allows NDSEQ version to **nondeterministically not prune** when pruning is possible.

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)
  atomic:
      if cost < best:
          best = cost
          best_soln = w
```

```
nondet-for (w in queue):
  if (lower_bnd(w) >= best):
      if (*): continue
  cost = compute_cost(w)

  if cost < best:
      best = cost
      best_soln = w
```

NDSEQ Specification

- **Claim:** NDSEQ code a good specification for the correctness of the parallelism.

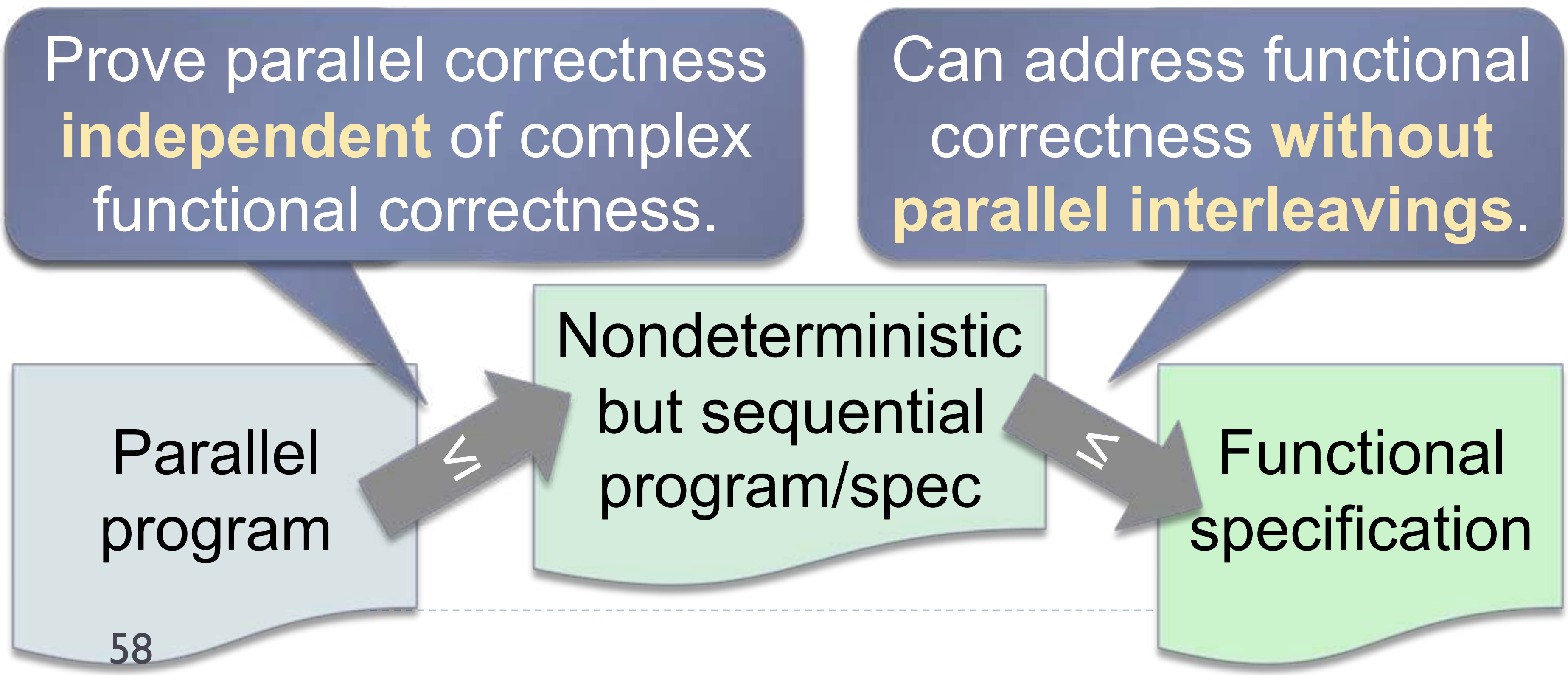
```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
      continue
  cost = compute_cost(w)
  atomic:
      if cost < best:
          best = cost
          best_soln = w
```

```
nondet-for (w in queue):
  if (lower_bnd(w) >= best):
      if (*): continue
  cost = compute_cost(w)

  if cost < best:
      best = cost
      best_soln = w
```

Recall: Our Approach

- Use **sequential but nondeterministic** specification for a program's parallelism.
  - User annotates **intended nondeterminism**.



NDSEQ Functional Correctness

- Claim: **much easier**
  - Consider recursive Boolean programs
  - Consider Model Checking: Reachability
  - Parallel Programs
    - pushdown system with multiple stacks
    - **Undecidable** [Ramalingam '00]
  - **Nondeterministic sequential** programs
    - pushdown systems
    - **Decidable** [Finkel et al. '97, Bouajjani et al. '97, and others]

Outline

- Overview
- Motivating Example
- Nondeterministic Sequential (NDSEQ) Specifications for Parallel Correctness
- **Proving Parallel Correctness**
- Future Work
- Conclusions

NDSEQ Specification

- Specifies:
  - For every parallel execution, there exists an NDSEQ execution with the same result.

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
     if (*): continue
  cost = compute_cost(w)
  atomic:
     if cost < best:
        best = cost
        best_soln = w
```

```
nondet-for (w in queue):
  if (lower_bnd(w) >= best):
     if (*): continue
  cost = compute_cost(w)
  atomic:
     if cost < best:
        best = cost
        best_soln = w
```

Proving NDSEQ Equivalence

- **Prove**: For every parallel execution, there is an NDSEQ one yielding the same result.

Parallel:

prune?(**a**)

prune?(**b**)

update(**b**)

prune?(**c**)

update(**a**)

best_soln: **(b)**

Proving NDSEQ Equivalence

- **Prove**: For every parallel execution, there is an NDSEQ one yielding the same result.

Parallel:	**NDSEQ:**
prune?(**a**)	prune?(**b**)
prune?(**b**)	update(**b**)
update(**b**)	prune?(**c**)
prune?(**c**)	prune?(**a**)
update(**a**)	update(**a**)
best_soln: **(b)**	best_soln: **(b)**

Proving NDSEQ Equivalence

- **Prove**: For every parallel execution, there is an NDSEQ one yielding the same result.

Parallel:	**NDSEQ:**
prune?(**a**)	prune?(**b**)
prune?(**b**)	update(**b**)
update(**b**)	prune?(**c**)
prune?(**c**)	prune?(**a**)
update(**a**)	update(**a**)
best_soln: **(b)**	best_soln: **(b)**

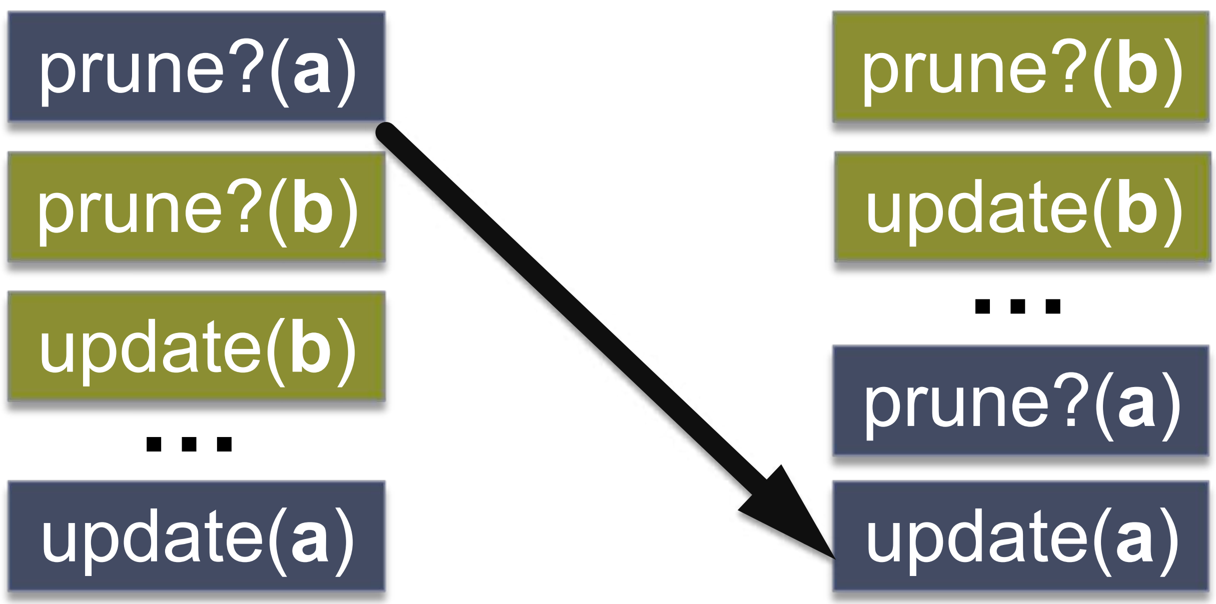
Proving NDSEQ Equivalence

Can we prove that such a rearrangement is always possible?

Parallel:	**NDSEQ:**
prune?(**a**)	prune?(**b**)
prune?(**b**)	update(**b**)
update(**b**)	prune?(**c**)
prune?(**c**)	prune?(**a**)
update(**a**)	update(**a**)
best_soln: **(b)**	best_soln: **(b)**

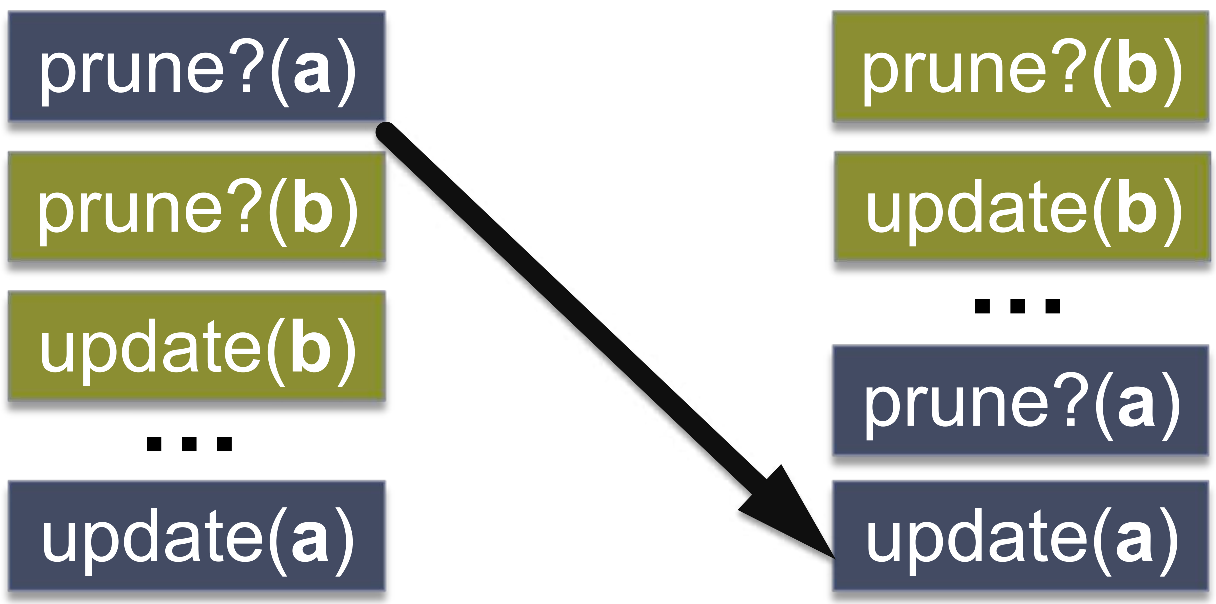
Proving NDSEQ Equivalence

- Is it always possible to move a **prune?** check later in a parallel execution without changing the result?



Proving NDSEQ Equivalence

- Is it always possible to move a **prune?** check later in a parallel execution without changing the result?
  - **Yes** – if the check **does not** prune.



Proving NDSEQ Equivalence

- (1) Can **prune?(x)** move past **prune?(y)**.

state: σ_1

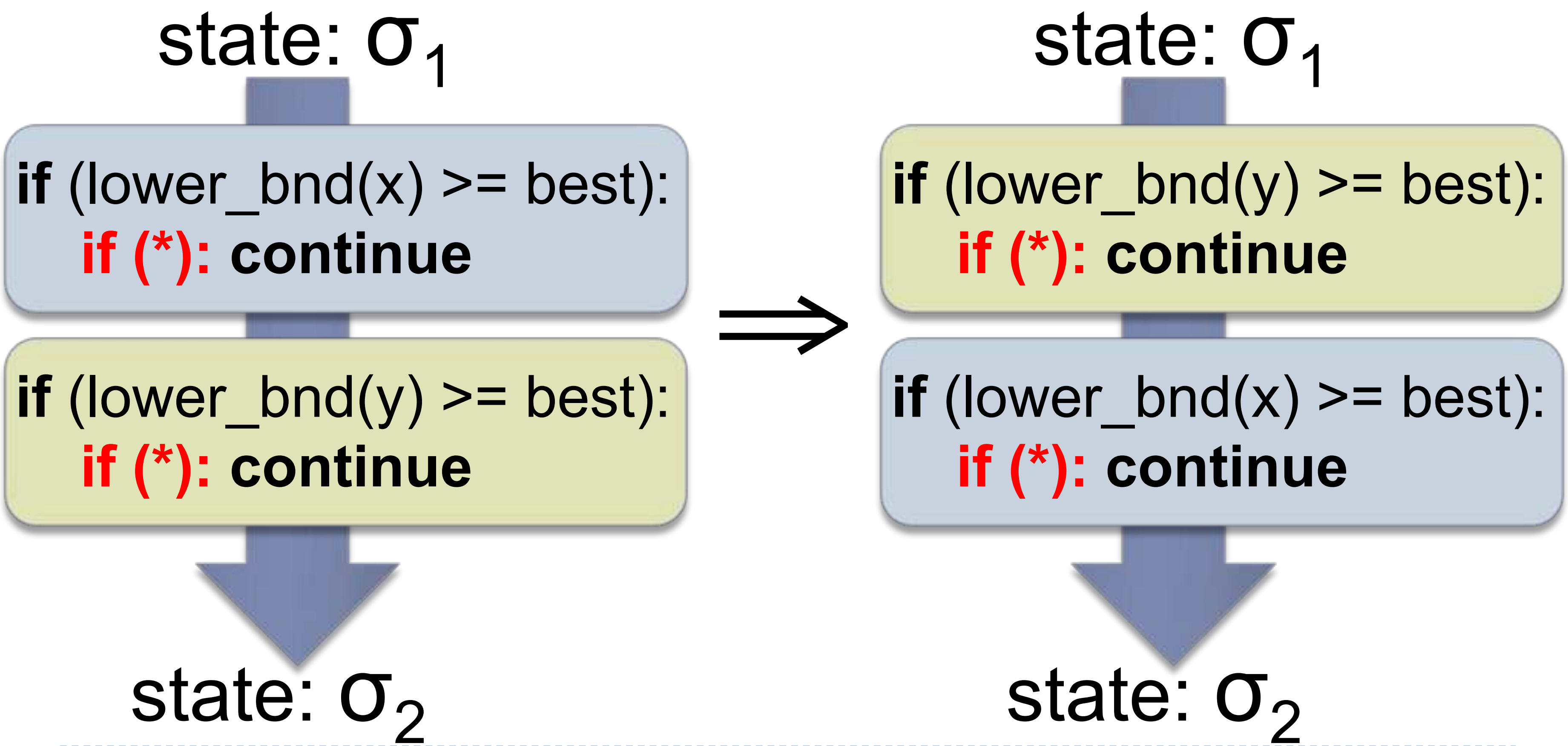
```
if (lower_bnd(x) >= best):
    if (*): continue
```

```
if (lower_bnd(y) >= best):
    if (*): continue
```

state: σ_2

Proving NDSEQ Equivalence

- (1) Can **prune?(x)** move past **prune?(y)**.



Proving NDSEQ Equivalence

- (2) Can **prune?(x)** move past **update?(y)**.

state: σ_1

if (lower_bnd(x) >= best):
    if (*): continue

best = *
best_soln = *

state: σ_2

Proving NDSEQ Equivalence

- (2) Can **prune?(x)** move past **update?(y)**.

state: σ_1

```
if (lower_bnd(x) >= best):
    if (*): continue
```

```
    best = *
    best_soln = *
```

state: σ_2

$\Rightarrow$

state: σ_1

```
    best = *
    best_soln = *
```

```
if (lower_bnd(x) >= best):
    if (*): continue
```

state: σ_2

Proving NDSEQ Equivalence

- This is proof by **reduction** [Lipton '75].
  - [Elmas, et al., POPL 09] has proved atomicity by reduction with SMT solvers.

```
parallel-for (w in queue):
  if (lower_bnd(w) >= best):
    if (*): continue
  cost = compute_cost(w)
  atomic:
    if cost < best:
      best = cost
      best_soln = w
```

Right-mover

Atomic

Outline

- Overview
- Motivating Example
- Nondeterministic Sequential (NDSEQ) Specifications for Parallel Correctness
- Proving Parallel Correctness
- **Future Work + Conclusions**

Future Work

- Prove parallel-NDSEQ equivalence for real benchmarks.
  - Automated proofs using SMT solving.
- Combine with tools for verifying sequential programs with nondeterminism.
  - Model checking techniques (e.g., CEGAR)
- Also interested in **dynamically** checking NDSEQ specifications.

NDSEQ and Debugging

- Given parallel execution exhibiting error:
  - Can we produce an NDSEQ trace exhibiting the same wrong behavior?
  - If so, bug is sequential and programmer can debug on a sequential (but NDSEQ) trace.
  - Can we efficiently produce NDSEQ trace given static proof of parallel correctness?
- Dynamically checking NDSEQ specs?
  - Ideally, efficiently: (1) finds equivalent NDSEQ trace, or (2) localizes parallel bug.

Questions?

Email jburnim@cs.berkeley.edu