Constraint Programming

Marco Kuhlmann & Guido Tack

Lecture 4

Today:
Constraint Propagation

Constraint propagation is a form of inference, not search, and as such is more "satisfying", both technically and aesthetically.

E.C. Freuder, 2005.

Brief recap:
A formal model for CP

Several levels

CSP

first-order logic

propagators and stores

Gecode/J programs

Several levels

CSP

propagators and stores

CSPs

- A **constraint satisfaction problem** is a triple (V,D,C) with
  - V: a set of variables
  - D: a finite domain
  - C: a set of constraints over V and D
- A **solution** of a CSP is a variable assignment that satisfies all constraints

CSPs

- This representation is **big**:
  - Each constraint is represented in extension

    $\Rightarrow$ possibly exponential size

  - Conjunction = intersection

    $\Rightarrow$ possibly exponential size

From CSPs to Models and Stores

- **CSP:**

  exponential *representation*

  - good for theoretical considerations
  - not implementable

From CSPs to Models and Stores

- **CSP:**

  exponential *representation*

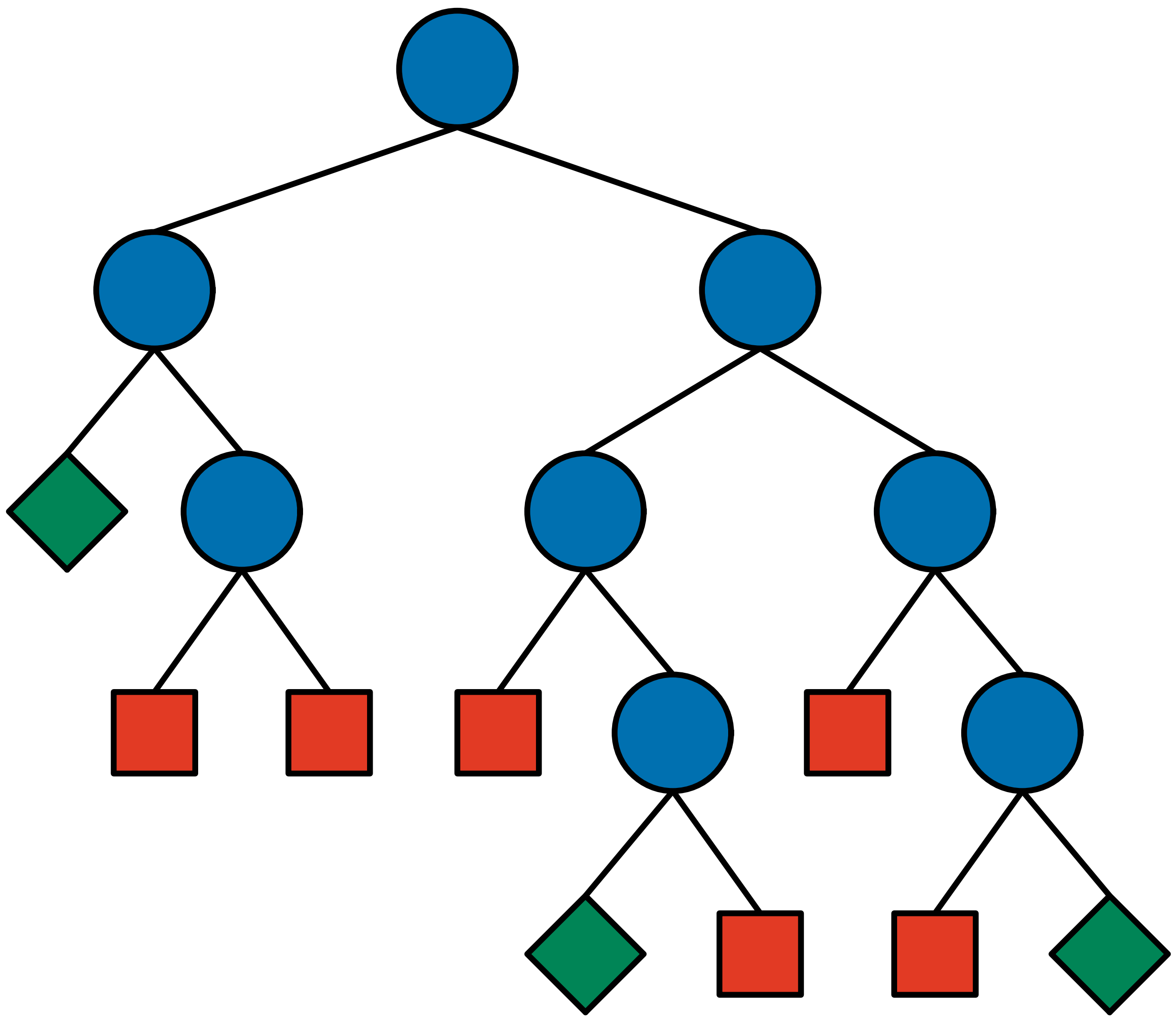
  - good for theoretical considerations
  - not implementable

- **Model / Store:**

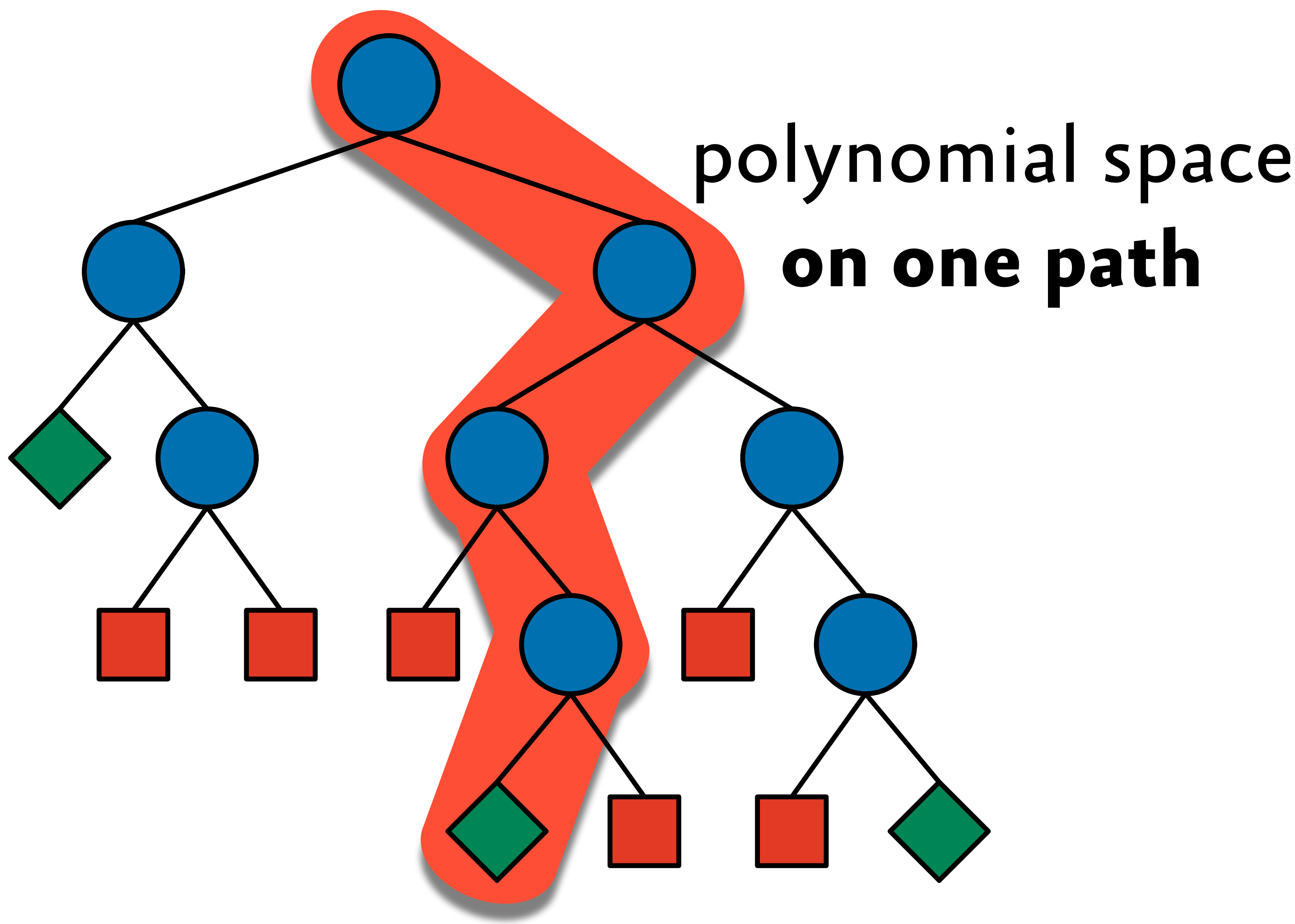
  exponential *computation*

  - close to an implementation
  - still formal enough to reason about it

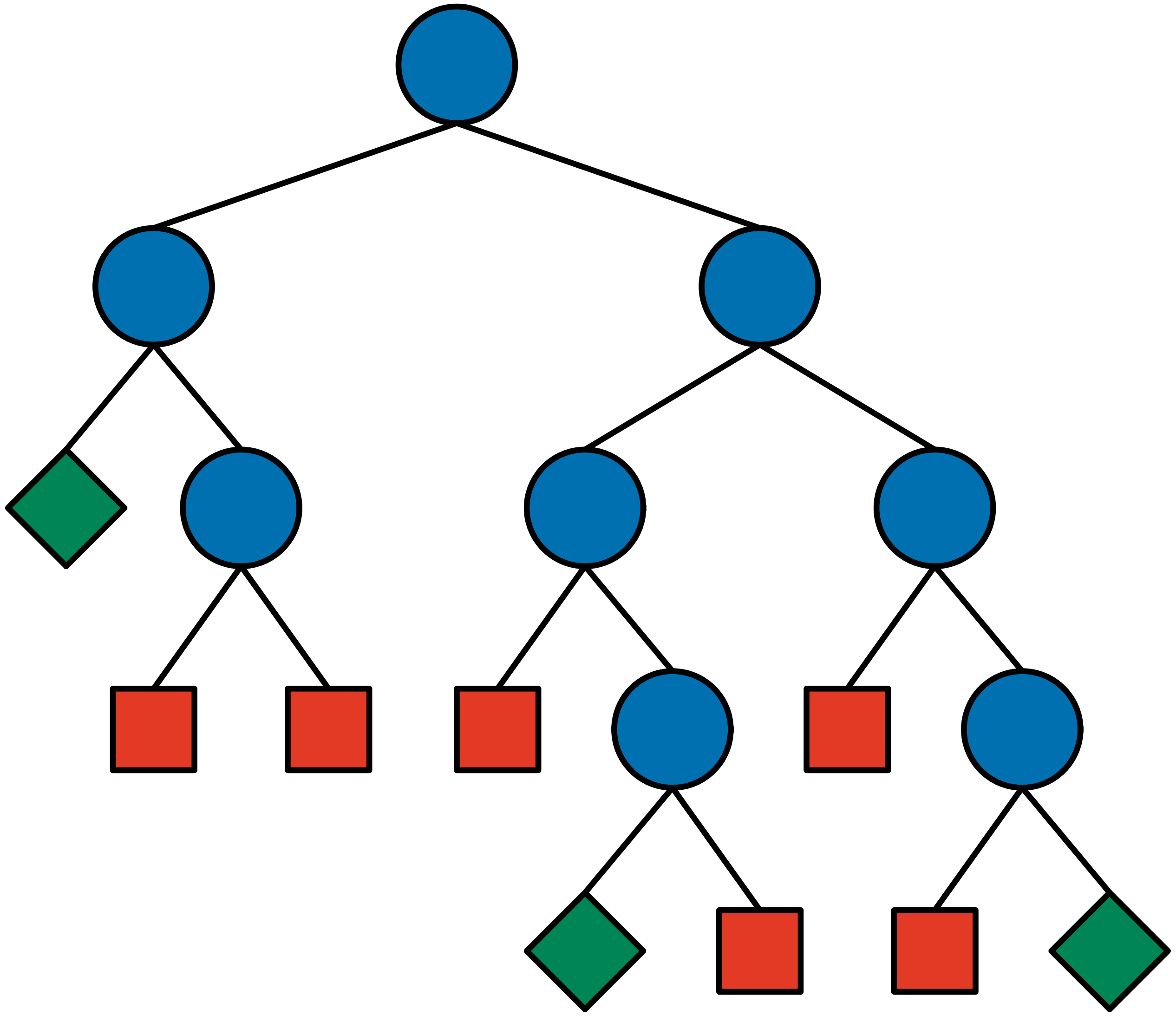
Trading space for time



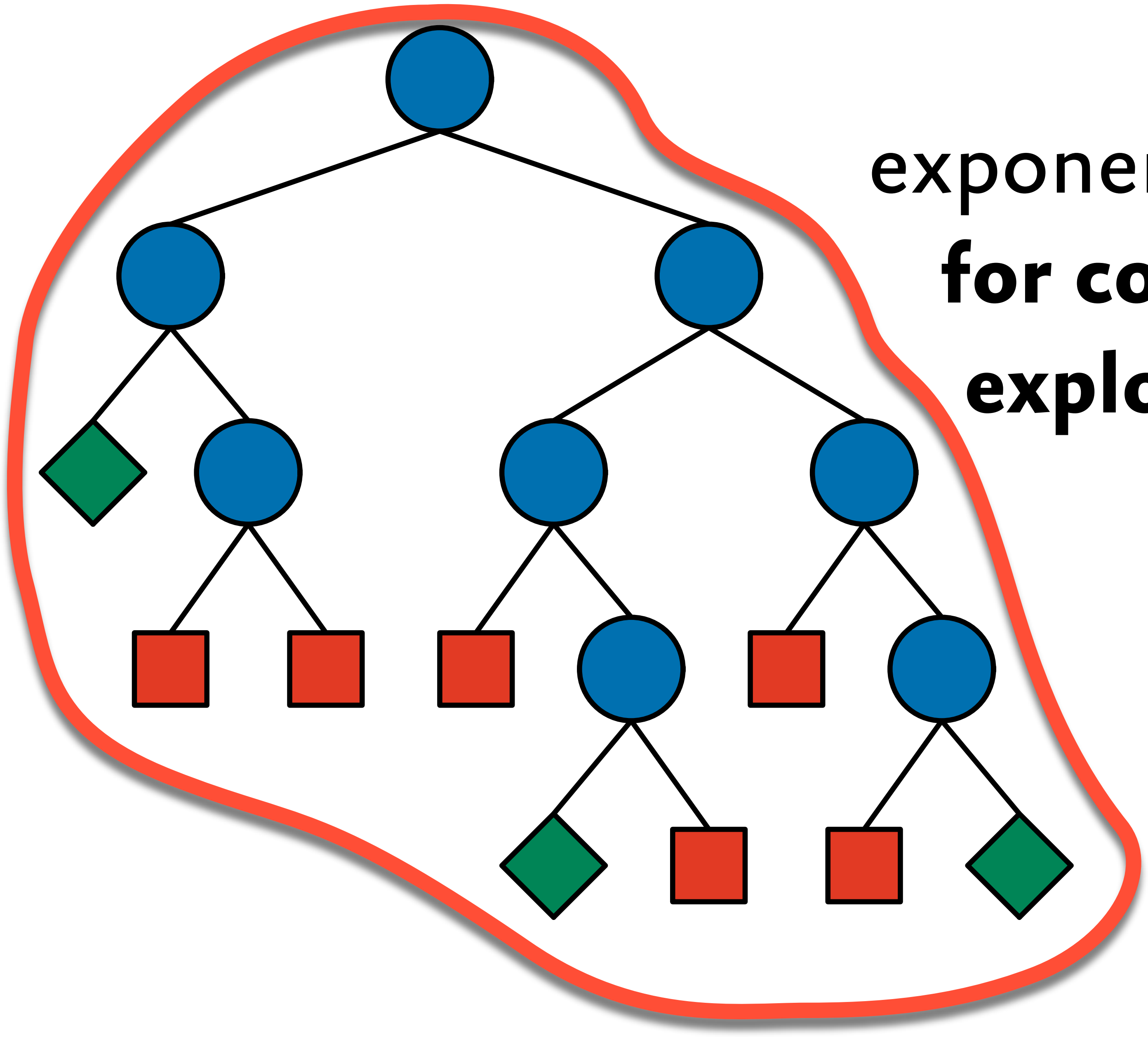
Trading space for time



Trading space for time



Trading space for time



exponential time **for complete exploration**

Models

- A **model** is a tuple (V, D, P, *b*) with
  - V, D: *variables* and *domain* as in CSPs
  - P: a set of *propagators*
  - *b:* a *branching*
- *Model* is the set of all models
- **We know how to implement functions!**

Stores

- A **store** captures *basic constraints*
- *Store* = $V \rightarrow 2^D$, mapping from variables to sets of values
- Propagators and branchings **operate on stores**
- *Store* ⊆ *Con*!   (slightly abusing notation)
- **The only constraints we represent explicitly!**

Propagators and branchings

- A propagator is a contracting, monotonic function

  $p \in \text{Store} \rightarrow \text{Store}$

- A branching is a function

  $b \in \text{Store} \rightarrow \text{Store} \times \text{Store}$

  such that

  $b(s).1 < s$ and $b(s).2 < s$ and

  $b(s).1 \cup b(s).2 = s$

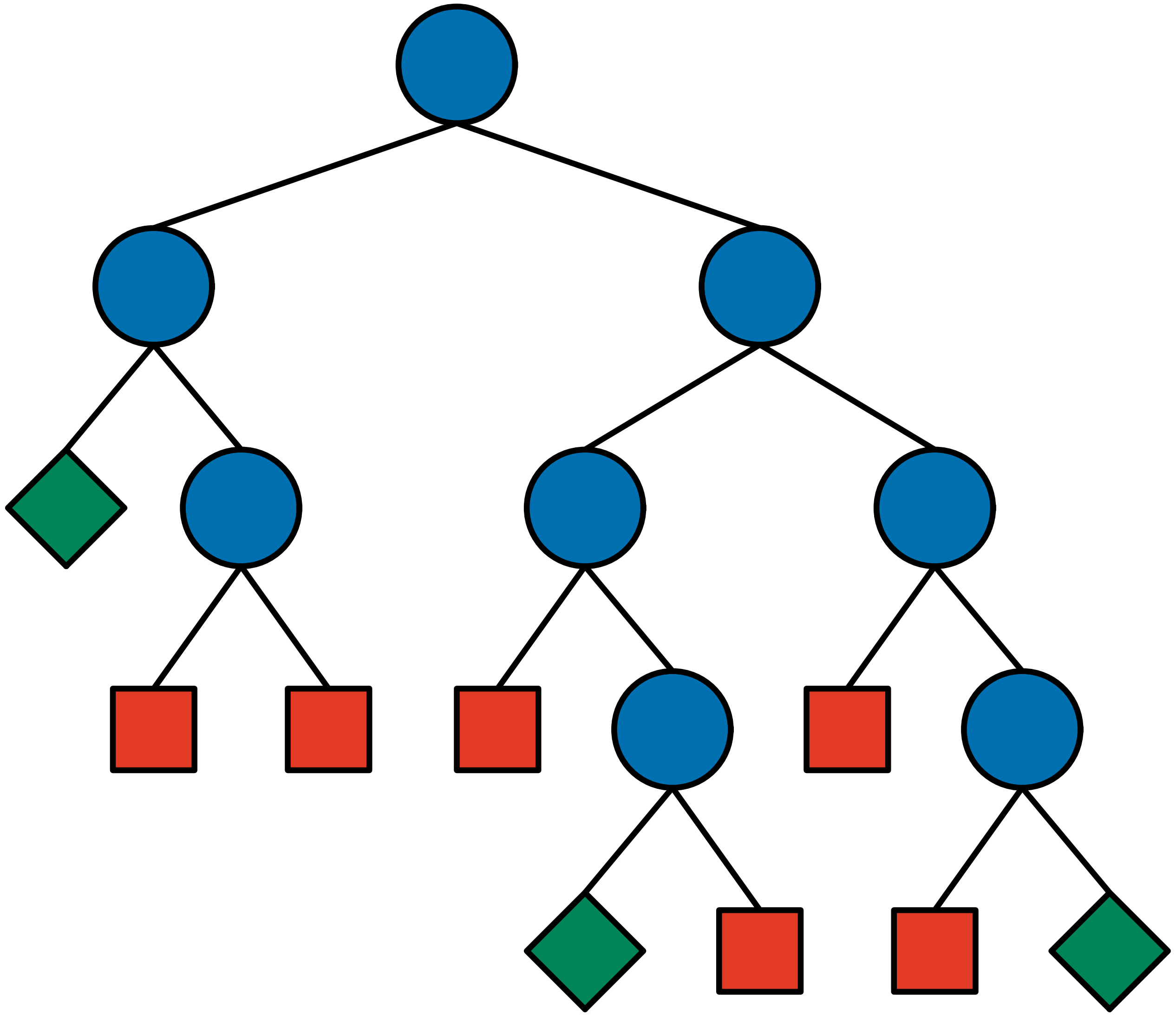
Solutions of models and stores

The **set of solutions** is defined as

$$sol((V, D, P, b), s) =$$
$$\{\alpha \mid \; store(\alpha) \subseteq S \wedge$$
$$\forall p \in P : p(store(\alpha)) = store(\alpha)\}$$

(all assignments licensed by the store and accepted by all propagators)

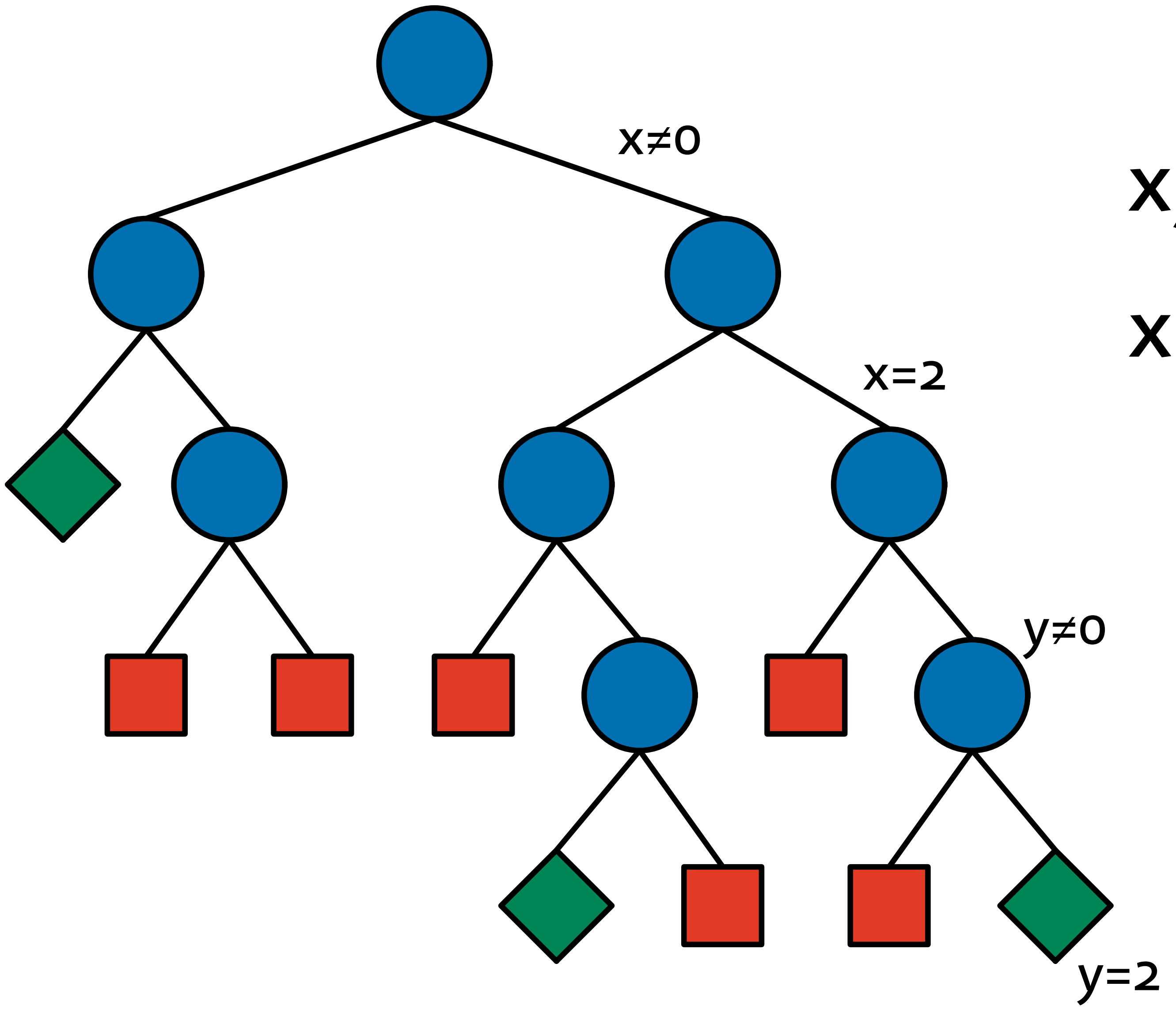
Generate and test



$x, y \in \{0,1,2\}$

$x = y$

Generate and test

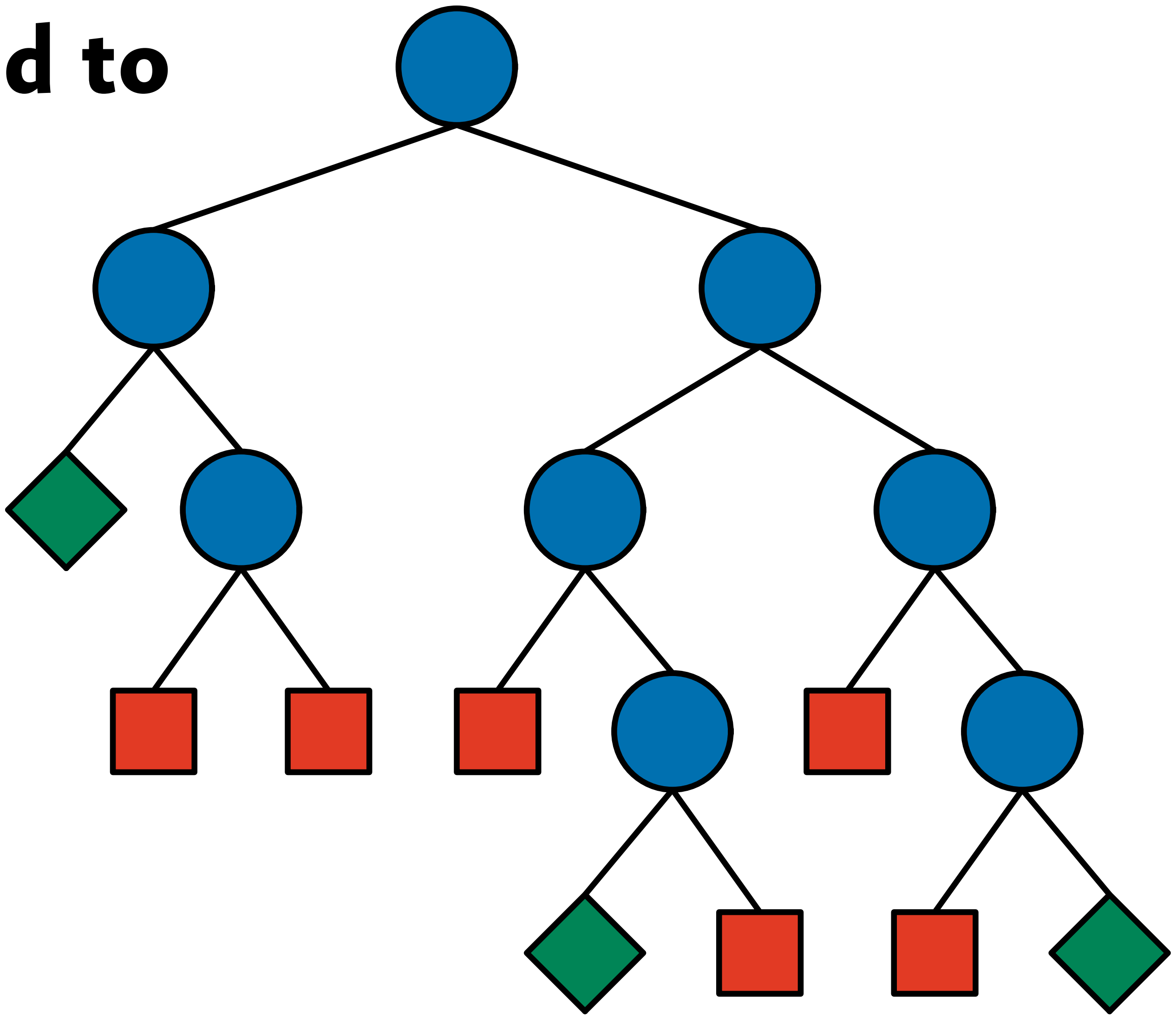


$x, y \in \{0,1,2\}$

$x = y$

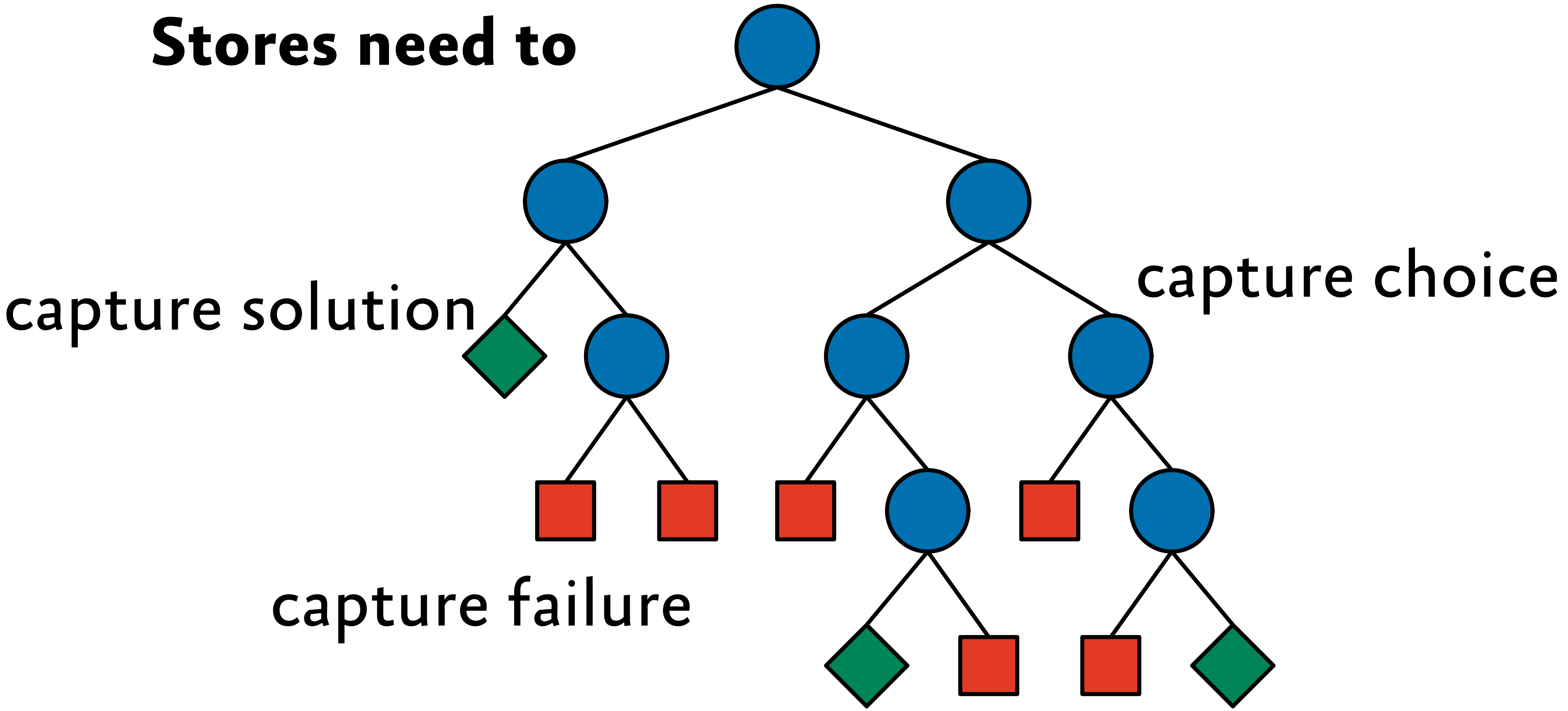
Generate and test

Stores need to

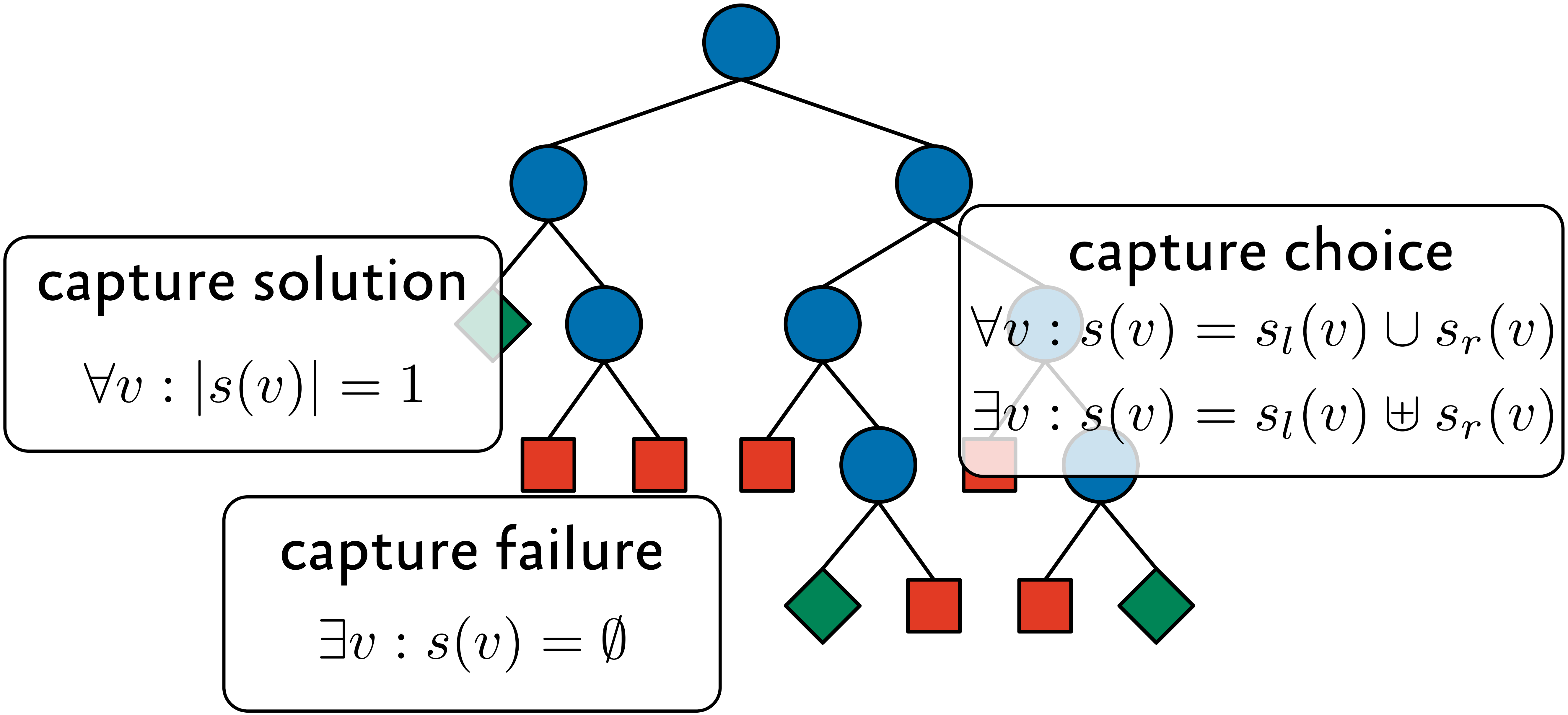


Generate and test

Stores need to



Generate and test



Generate and test

capture solution

$$\forall v : |s(v)| = 1$$

capture failure

$$\exists v : s(v) = \emptyset$$

capture choice

$$\forall v : s(v) = s_l(v) \cup s_r(v)$$

$$\exists v : s(v) = s_l(v) \uplus s_r(v)$$

Generate and test

capture solution

$$\forall v : |s(v)| = 1$$

capture failure

$$\exists v : s(v) = \emptyset$$

capture choice

$$\forall v : s(v) = s_l(v) \cup s_r(v)$$

$$\exists v : s(v) = s_l(v) \uplus s_r(v)$$

Store = V → 2^{D}

expressive enough!

Generate and test

capture solution

$$\forall v : |s(v)| = 1$$

capture failure

$$\exists v : s(v) = \emptyset$$

capture choice

$$\forall v : s(v) = s_l(v) \cup s_r(v)$$

$$\exists v : s(v) = s_l(v) \uplus s_r(v)$$

propagator:

p_C implements C:

$\alpha \in C \Leftrightarrow p_C(store(\alpha)) = store(\alpha)$

branching:

$b(s) = (s_l, s_r)$

Generate and test

capture solution

$$\forall v : |s(v)| = 1$$

capture failure

$$\exists v : s(v) = \emptyset$$

propagator:

p_C implements C:

$$\alpha \in C \Leftrightarrow p_C(store(\alpha)) = store(\alpha)$$

use propagators for checking!

capture choice

$$\forall v : s(v) = s_l(v) \cup s_r(v)$$

$$\exists v : s(v) = s_l(v) \uplus s_r(v)$$

branching:

$$b(s) = (s_l, s_r)$$

branchings generate assignments!

Generate and test

```
gt( (V,D,P,b), s)

  if s not singleton
    (sₗ,sᵣ) := b(s)
    return gt( (V,D,P,b), sₗ) or
           gt( (V,D,P,b), sᵣ)
```

Generate and test

```
gt( (V,D,P,b), s)

  if s not singleton
    (s_l,s_r) := b(s)
    return gt( (V,D,P,b), s_l) or
           gt( (V,D,P,b), s_r)
```

use **branching** to generate

Generate and test

```
gt( (V,D,P,b), s)

  if s not singleton
    (s_l,s_r) := b(s)
    return gt( (V,D,P,b), s_l) or
           gt( (V,D,P,b), s_r)
```

use **branching** to generate

search recursively

Generate and test

```
gt( (V,D,P,b), s)

  if s not singleton
    (sₗ,sᵣ) := b(s)
    return gt( (V,D,P,b), sₗ) or
           gt( (V,D,P,b), sᵣ)
  else
    forall p∈P
      if p(s) is failed return false;
    return true;
```

use **branching** to generate

search recursively

use **propagators** to test

Generate and test

```
gt( (V,D,P,b), s)

  if s not singleton
    (sₗ,sᵣ) := b(s)
    return gt( (V,D,P,b), sₗ) or
           gt( (V,D,P,b), sᵣ)
  else
    forall p∈P
      if p(s) is failed return false;
    return true;
```

partition search space

search exhaustively

implement constraints

Generate and test

```
gt( (V,D,P,b), s)

  if s not singleton
    (sₗ,sᵣ) := b(s)
    return gt( (V,D,P,b), sₗ) or
           gt( (V,D,P,b), sᵣ)
  else
    forall p∈P
      if p(s) is failed return false;
    return true;
```

complete-
ness

correctness

Towards propagation

```
gt( (V,D,P,b), s)

  if s not singleton
    (sₗ,sᵣ) := b(s)
    return gt( (V,D,P,b), sₗ) or
           gt( (V,D,P,b), sᵣ)
  else
    forall p∈P
      if p(s) is failed return false;
    return true;
```

use **propagators** to test

Towards propagation

```
gt( (V,D,P,b), s)

  if s not singleton
    (sₗ,sᵣ) := b(s)
    return gt( (V,D,P,b), sₗ) or
           gt( (V,D,P,b), sᵣ)
  else
    forall p∈P
      if p(s) is failed return false;
    return true;
```

use **propagators** to …

Towards propagation

propagate!

```
gt( (V,D,P,b), s)
  s' := propagate( (V,D,P,b), s)
  if s' not singleton
    (s_l,s_r) := b(s')
    return gt( (V,D,P,b), s_l) or
           gt( (V,D,P,b), s_r)
  else

      if s' is failed return false;
    return true;
```

Naive constraint propagation

Preliminaries: Well-founded order

- A strict partial order $(S, <)$ is **well-founded** iff no infinite sequence $s_1, s_2, s_3, \ldots$ with $s_i \in S$ exists s.th. $x_{i+1} < x_i$
- Examples: $(\mathbb{N}, <)$, $(2^X, \subset)$ and $(\mathrm{Store}, <)$

Preliminaries: Lexicographic order

- For two partial orders $(X,\leq_x)$ and $(Y,\leq_y)$, the **lexicographic order** $(X{\times}Y, \leq_{lex})$ is defined as

  $(x_1,y_1) \leq_{lex} (x_2,y_2)$ ⇔ $x_1 \leq_x x_2$ and $x_1{\neq}x_2$ or

  $x_1{=}x_2$ and $y_1 \leq_y y_2$

- Well-founded, if $(X,\leq_x)$ and $(Y,\leq_y)$ are well-founded

Preliminaries: Fixpoint

- For a function $f \in X \rightarrow X$

  $x \in X$ is a **fixpoint** of f iff

  $$f(x) = x$$

Naive constraint propagation

- We are looking for a function

  propagate: $\mathit{Model} \times \mathit{Store} \rightarrow \mathit{Store}$

- Starting from an initial store
- Returning store where all possible constraint propagation has been performed
- For now: focus on the basic idea

Naive propagation function

```
propagate ( (V,D,P,b), s)
  while p∈P and p(s)≠s do
     s := p(s);
   return s;
```

- **Questions:**
  - Does it terminate?
  - What does it compute?

Naive propagation: termination

```
propagate ( (V,D,P,b), s)
  while p∈P and p(s)≠s do
    s := p(s);
  return s;
```

- Consider store s_i at iteration *i*:

  $s_{i+1} < s_i$

- As (Store,<) is well-founded, the loop terminates

Naive propagation: result

- For `propagate(M,s) = s'`, we can show
  - sol (M, s) = sol (M, s')
  - for all p∈prop(M): p(s') = s'

Naive propagation: result

- For `propagate(M,s) = s'`, we can show

no solution removed

  - sol (M, s) = sol (M, s')
  - for all p∈prop(M): p(s') = s'

Naive propagation: result

- For `propagate(M,s) = s'`, we can show
  - sol (M, s) = sol (M, s')
  - for all p∈prop(M): p(s') = s'

no solution removed

largest simultaneous fixpoint

Fixpoint

- Assume `propagate( (V,D,P,b),s) = s'`

  Then s' is the largest simultaneos fixpoint of P with $s' \leq s$.
  That means:

  - for all $p \in P$: $p(s') = s'$ (clear from termination)
  - any other fixpoint is smaller (proof needed!)

```
propagate ( (V,D,P,b), s)
 while p∈P and p(s)≠s do
   s := p(s);
 return s;
```

Fixpoint

- Assume `propagate( (V,D,P,b),s) = s'`

  Then s' is the largest simultaneos fixpoint of P with s'≤s.
  That means:

  - for all p∈P: p(s') = s' (clear from termination)
  - any other fixpoint is smaller (proof needed!)

Fixpoint

- Assume `propagate( (V,D,P,b),s) = s'`

  Then s' is the largest simultaneos fixpoint of P with s'≤s.
  That means:

  - for all p∈P: p(s') = s' (clear from termination)
  - any other fixpoint is smaller (proof needed!)

Largest fixpoint

- Let p_i be the propagator of the i-th iteration

  $s_i := p_i (s_{i-1})$ for $i > 0$, $s_0 = s$

- Loop terminates after *n* iterations with s_n
- Assume t is simultaneous fixpoint with $t \leq s$
- Show $t \leq s_n$
- Prove by induction over *i* that $t \leq s_i$

```
propagate ( (V,D,P,b), s)
 while p∈P and p(s)≠s do
   s := p(s);
  return s;
```

Largest fixpoint

- Let p_i be the propagator of the i-th iteration

  $s_i := p_i (s_{i-1})$ for $i > 0$, $s_0 = s$

- Loop terminates after *n* iterations with s_n
- Assume t is simultaneous fixpoint with $t \leq s$
- Show $t \leq s_n$
- Prove by induction over *i* that $t \leq s_i$

Largest fixpoint

- Let p_i be the propagator of the i-th iteration

  $s_i := p_i(s_{i-1})$ for $i > 0$, $s_0 = s$

- Loop terminates after *n* iterations with s_n
- Assume t is simultaneous fixpoint with $t \leq s$
- Show $t \leq s_n$
- Prove by induction over *i* that $t \leq s_i$

Largest fixpoint: base case

For i=0:

$t \leq s_0$ because $s_0 = s$ and we assumed $t \leq s$

Largest fixpoint: induction step

From i to i+1:

$t \leq s_i$

$\Rightarrow p_{i+1}(t) \leq p_{i+1}(s_i)$ — p_{i+1} monotonic

$\Rightarrow t = p_{i+1}(t) \leq p_{i+1}(s_i)$ — t is fixpoint of p_{i+1}

$\Rightarrow t \leq p_{i+1}(s_i) = s_{i+1}$ — definition of s_i

$\Rightarrow t \leq s_{i+1}$

What makes this naive?

- Termination relies on strict contraction
- We always have to check all propagators for one that can strictly contract

Ideas:

- Maintain propagators which are known to be at fixpoint
- Look at the variables that propagators actually compute with

⇒ Dependency-directed propagation

```
propagate ( (V,D,P,b), s)
  while p∈P and p(s)≠s do
    s := p(s);
  return s;
```

What makes this naive?

- Termination relies on strict contraction
- We always have to check all propagators for one that can strictly contract

Ideas:

- Maintain propagators which are known to be at fixpoint
- Look at the variables that propagators actually compute with

⇒ Dependency-directed propagation

What makes this naive?

- Termination relies on strict contraction
- We always have to check all propagators for one that can strictly contract

Ideas:

- Maintain propagators which are known to be at fixpoint
- Look at the variables that propagators actually compute with

⇒ Dependency-directed propagation

Realistic constraint propagation

Ideas for improving propagation

- Propagator narrows only some domains
  - re-propagate only propagators that "care about" the changed variables
- Maintain a set of "dirty" propagators
  - dirty = possibly not at fixpoint for current store
  - all "clean" propagators known to be at fixpoint
  - only propagate dirty propagators

Scope of a propagator

- *scope(p)*: variables that the propagator cares about
- for all variables *outside* the scope of *p*:
  - *p* does not consider their domain for propagation (no input)
  - *p* does not narrow their domain (no output)

Dependency-directed propagation

- maintain a set DP of "dirty" propagators
- chose next propagator from DP instead of P
- when run, remove propagator from DP
- compute changed variables CV
- add all p' with $scope(p') \cap CV \neq \emptyset$ to DP
- note: this may add p again!

Improved propagation

```
propagate ( (V,D,P,b), s0)
  s := s0; DP = P;
  while DP ≠ ∅ do
    choose p∈DP;
    s' := p(s); DP = DP−{p};
    MV := { x∈V | s(x)≠s'(x) };
    N  := { q∈P | ∃x∈var(q): x∈MV };
    DP := DP ∪ N;
    s := s';
  return s;
```

Improved propagation

- Does it still compute the largest sim. fixpoint?
  - Prove using loop invariant
- Does it terminate?
  - not trivial any more, as possibly $s_{i+1} = s_i$!

Loop invariant

- The loop has the following invariant:

  for all $p \in P - DP$: $p(s) = s$

- After termination, we have $DP = \varnothing$, so

  for all $p \in P$: $p(s) = s$

- Proof obligations:
  - invariant holds initially
  - invariant is invariant

Loop invariant

- Holds initially, as P−DP=∅ (initialization: DP := P)
- Is invariant:

```
s' := p(s); DP = DP−{p};
MV := { x∈V | s(x)≠s'(x) };
N  := { q∈P | ∃x∈var(q): x∈MV };
DP := DP ∪ N;
```

re-add propagators with modified variables

Improved propagation - fixpoint

- Loop invariant guarantees fixpoint
- As for naive propagation, it is the largest simultaneous fixpoint
  - proof for naive version still works here

Improved propagation - termination

- Insight:
  - if $MV=\varnothing$, then p is removed from DP
  - if $MV\neq\varnothing$, then $p(s) < s$
- Consider pairs (s_i, DP_i) with
  - s_i the store at the i-th iteration
  - DP_i the set DP at the i-th iteration
- Strictly decreasing w.r.t. well-founded lexicographic order of (Store,<) and $(2^P, \subset)$

Further improvements

Using fixpoint knowledge

- Up to now:

  to find out whether p is at fixpoint, we have to propagate p!

- Idea:

  let the propagator provide information about whether it is at fixpoint

Subsumed propagators

- A propagator is **subsumed** by a store s, iff

  for all $s' \leq s$: $p(s') = s'$
- **All stronger stores are fixpoints**
- (p is entailed by s, s entails p, s subsumes p)
- Remove p from P! Not needed **from now on**

Subsumed propagator: example

- Consider the propagator $p_{\leq}$ for $x \leq y$:

$$p_{\leq}(s) = \{\, x \to \{\, n \in s(x) \mid n \leq \max(s(y))\},\; y \to \{\, n \in s(y) \mid n \geq \min(s(x))\}\}$$

- $p_{\leq}$ is entailed by $s = \{\, x \to \{1,2,3\},\, y \to \{3,4,5\}\,\}$

$$p_{\leq}(s) = \{\, x \rightarrow \{\, n \in s(x) \mid n \leq \max(s(y))\},$$
$$y \rightarrow \{\, n \in s(y) \mid n \geq \min(s(x))\}\}$$

Fixpoints

- Let us look at $p_{\leq}$ again
- After executing $p_{\leq}$, we can show that it is at fixpoint!
- But: $var(p_{\leq})=\{x,y\}$, so we add $p_{\leq}$ to DP
- How can we avoid that?

First idea: idempotent functions

- A function $f \in X \rightarrow X$ is **idempotent** iff

  for all $x \in X$: $f(f(x)) = f(x)$

- For propagators:
  - p(p(s)) = p(s), **for all stores!**
  - very strong property!
  - (but required in some CP systems, e.g. Mozart)

Better: weak idempotence

- A function $f \in X \to X$ is **idempotent** on $x \in X$ iff

  $f(f(x)) = f(x)$ now for just one element!

- For propagators, this means

  if p is idempotent on s, it is not necessarily idempotent on s' with s'≤s

How to find out?

- Propagator returns status message

  $p \in Store \to SM \times Store$

  with SM = {fix, nofix, subsumed}

- p(s) = (fix, s'): s' is fixpoint for p
- p(s) = (subsumed, s'): s' subsumes p
- p(s) = (nofix, s'): possibly no fixpoint, as before

Extend propagation function

```
propagate ( (V,D,P,b), s₀)
  s := s₀; DP = P;
  while DP ≠ ∅ do
    choose p∈DP;
    (stat,s') := p(s); DP = DP−{p};
    if stat=subsumed then P:=P−{p};
    MV := { x∈V | s(x)≠s'(x) };
    N  := { q∈P | ∃x∈var(q): x∈MV };
    if stat=fix then N:=N−{p};
    DP := DP ∪ N;
    s := s';
  return (P,s);
```

Extend propagation function

```
propagate ( (V,D,P,b), s₀)
  s := s₀; DP = P;
  while DP ≠ ∅ do
    choose p∈DP;
    (stat,s') := p(s); DP = DP−{p};
    if stat=subsumed then P:=P−{p};
    MV := { x∈V | s(x)≠s'(x) };
    N  := { q∈P | ∃x∈var(q): x∈MV };
    if stat=fix then N:=N−{p};
    DP := DP ∪ N;
    s := s';
  return (P,s);
```

Extend propagation function

```
propagate ( (V,D,P,b), s₀)
   s := s₀; DP = P;
   while DP ≠ ∅ do
     choose p∈DP;
     (stat,s') := p(s); DP = DP−{p};
     if stat=subsumed then P:=P−{p};
     MV := { x∈V | s(x)≠s'(x) };
     N  := { q∈P | ∃x∈var(q): x∈MV };
     if stat=fix then N:=N−{p};
     DP := DP ∪ N;
     s := s';
   return (P,s);
```

Extend propagation function

```
propagate ( (V,D,P,b), s0)
  s := s0; DP = P;
  while DP ≠ ∅ do
    choose p∈DP;
    (stat,s') := p(s); DP = DP−{p};
    if stat=subsumed then P:=P−{p};
    MV := { x∈V | s(x)≠s'(x) };
    N  := { q∈P | ∃x∈var(q): x∈MV };
    if stat=fix then N:=N−{p};
    DP := DP ∪ N;
    s := s';
  return (P,s);
```

Extend propagation function

```
propagate ( (V,D,P,b), s0)
  s := s0; DP = P;
  while DP ≠ ∅ do
    choose p∈DP;
    (stat,s') := p(s); DP = DP−{p};
    if stat=subsumed then P:=P−{p};
    MV := { x∈V | s(x)≠s'(x) };
    N  := { q∈P | ∃x∈var(q): x∈MV };
    if stat=fix then N:=N−{p};
    DP := DP ∪ N;
    s := s';
  return (P,s);
```

Correctness

- **We have to check that**
  - the invariant is still invariant
  - all solutions are preserved
  - it still computes the largest simultaneous fixpoint
- **Argument: fixpoints!**

Propagation events

- For many propagators, we can easily decide whether still at fixpoint when domain changes
- We need to know *how* the domain changed
- Describe by **propagation event** (or just event)

Events: example

- Take the propagator $p_{\leq}$ again as an example

$$p_{\leq}(s) = \{ x \rightarrow \{ n \in s(x) \mid n \leq \max(s(y))\}, \\ y \rightarrow \{ n \in s(y) \mid n \geq \min(s(x))\}\}$$

- Only propagate if max(s(y)) or min(s(x)) changes!

Events: another example

- Take the propagator $p_{\neq}$ as an example

  $$p_{\neq}(s) = \{ x \rightarrow s(x) - single(s(y)), \\ y \rightarrow s(y) - single(s(x))\}$$

  where $single(\{n\}) = \{n\}$, $single(X)=X$ otherwise

- Only propagate if x or y become assigned!

Events

- Typical events:
  - fix(x) — x is assigned
  - min(x) — minimum of x changed
  - max(x) — maximum of x changed
  - any(x) — domain of x changed
- Clearly overlap:
  - fix(x) implies any(x) and min(x) or max(x)
  - min(x) or max(x) imply any(x)

Computing events

- When the store changes, for s'≤s:

$$
\begin{aligned}
events(s, s') \quad = \quad & \{\mathrm{any}(x) \mid s'(x) \subset s(x)\} \cup \\
& \{\min(x) \mid \min s'(x) > \min s(x)\} \cup \\
& \{\max(x) \mid \max s'(x) < \max s(x)\} \cup \\
& \{\mathrm{fix}(x) \mid |s'(x)| = 1 \wedge |s(x)| > 1\}
\end{aligned}
$$

- Events are monotonic:

$$s'' \leq s' \leq s : \ events(s, s'') = events(s, s') \cup events(s', s'')$$

Computing events: example

- Given two stores

$$
\begin{aligned}
s_1 &= \{x_1 \mapsto \{1,2,3\}, x_2 \mapsto \{3,4,5,6\}, x_3 \mapsto \{0,1\}, x_4 \mapsto \{7,8,10\}\} \\
s_2 &= \{x_1 \mapsto \{1,2\}, x_2 \mapsto \{3,5,6\}, x_3 \mapsto \{1\}, x_4 \mapsto \{7,8,10\}\}
\end{aligned}
$$

- Then

$$events(s_1, s_2) = \{\max(x_1), \text{any}(x1), \text{any}(x_2), \text{fix}(x_3), \min x_3, \text{any}(x_3)\}$$

Event sets for propagators

- Associate with every propagator p an event set es(p)
- Required properties:
  - es(p) must contain some events that occur between stores s and s', if s'≤s, p(s)=s, p(s')≠s
  - if p(p(s))≠p(s), then es(p) must occur some events from events(s, p(s))

Propagation with events

```
propagate ( (V,D,P,b), s₀)
   s := s₀; DP = P;
   while DP ≠ ∅ do
     choose p∈DP;
     (stat,s') := p(s); DP = DP−{p};
     if stat=subsumed then P:=P−{p};

     N  := { q∈P | events(s,s') ∩ es(q) ≠ ∅ };
     if stat=fix then N:=N−{p};
     DP := DP ∪ N;
     s := s';
   return (P,s);
```

Summary

CSPs, models and stores

- CSPs are abstract, mathematical objects
  - good for reasoning and proofs
  - not directly implementable
- Stores capture basic constraints
- Models containt propagators and branchings
  - propagators implement constraints on stores
  - branchings generate all assignments

Constraint propagation

- Propagators are contracting, monotonic functions on stores
- Compute largest simultaneous fixpoint of propagators
- Propagation preserves solutions
- Propagators are strong enough to decide for assignments

Efficient constraint propagation

- Dependency-directed propagation
  - only re-run propagators whose variables have changed
- Use fixpoint knowledge to avoid useless re-execution
  - idempotence, subsumption, events
  - knowledge is provided by the propagator

Pointers

- **Finite Domain Constraint Programming Systems**, Christian Schulte, Mats Carlsson. In: Handbook of CP, 2006.
- **Efficient Constraint Propagation Engines**, Christian Schulte, Peter J. Stuckey. *CoRR entry,* 2006.

Thank you for your attention.