Logic Programming (CTM 12.3-12.5)

Constraint Programming: Constraints and Computation Spaces

Carlos Varela

Rensselaer Polytechnic Institute

December 3, 2021

Generate and Test Example

- We can use the relational computation model to generate all digits:

```
fun {Digit}
   choice 0 [] 1 [] 2 [] 3 [] 4 [] 5 [] 6 [] 7 [] 8 [] 9 end
end
{Browse {Search.base.all Digit}}
% displays [0 1 2 3 4 5 6 7 8 9]
```

Finding palindromes

- Find all four-digit palindromes that are products of two-digit numbers:

```
fun {Palindrome}
   X in
   X = (10*{Digit}+{Digit})*(10*{Digit}+{Digit})      % generate
   (X>=1000) = true                                   % test
   (X div 1000) mod 10 = (X div 1) mod 10             % test
   (X div 100) mod 10 = (X div 10) mod 10             % test
   X
end
{Browse {Search.base.all Palindrome}}                 % 118 solutions
```

Propagate and Search

- The *generate and test* programming pattern can be very inefficient (e.g., Palindrome program explores 10000 possibilities).

- An alternative is to use a *propagate and search* technique.

  Propagate and search filters possibilities during the generation process, to prevent combinatorial explosion when possible.

Propagate and Search

Propagate and search approach is based on three key ideas:

•*Keep partial information*, e.g., "in any solution, X is greater than 100".

•*Use local deduction*, e.g., combining "X is less than Y" and "X is greater than 100", we can deduce "Y is greater than 101" (assuming Y is an integer.)

•*Do controlled search*. When no more deductions can be done, then search. Divide original CSP problem P into two new problems: (P ^ C) and (P ^ ¬C) and where C is a new constraint. The solution to P is the union of the two new sub-problems. Choice of C can significantly affect search space.

Propagate and Search Example

- Find two digits that add to 10, multiply to more than 24:

```
D1::0#9          D2::0#9                  % initial constraints
{Browse D1}      {Browse D2}              % partial results
D1+D2 =: 10  % reduces search space from 100 to 81 possibilities
                 % D1 and D2 cannot be 0.
D1*D2 >=: 24   % reduces search space to 9 possibilities
                 % D1 and D2 must be between 4 and 6.
D1 <: D2         % reduces search space to 4 possibilities
                 % D1 must be 4 or 5 and D2 must be 5 or 6.
                 % It does not find unique solution D1=4 and D2=6
```

Propagate and Search Example(2)

- Find a rectangle whose perimeter is 20, whose area is greater than or equal to 24, and width less than height:

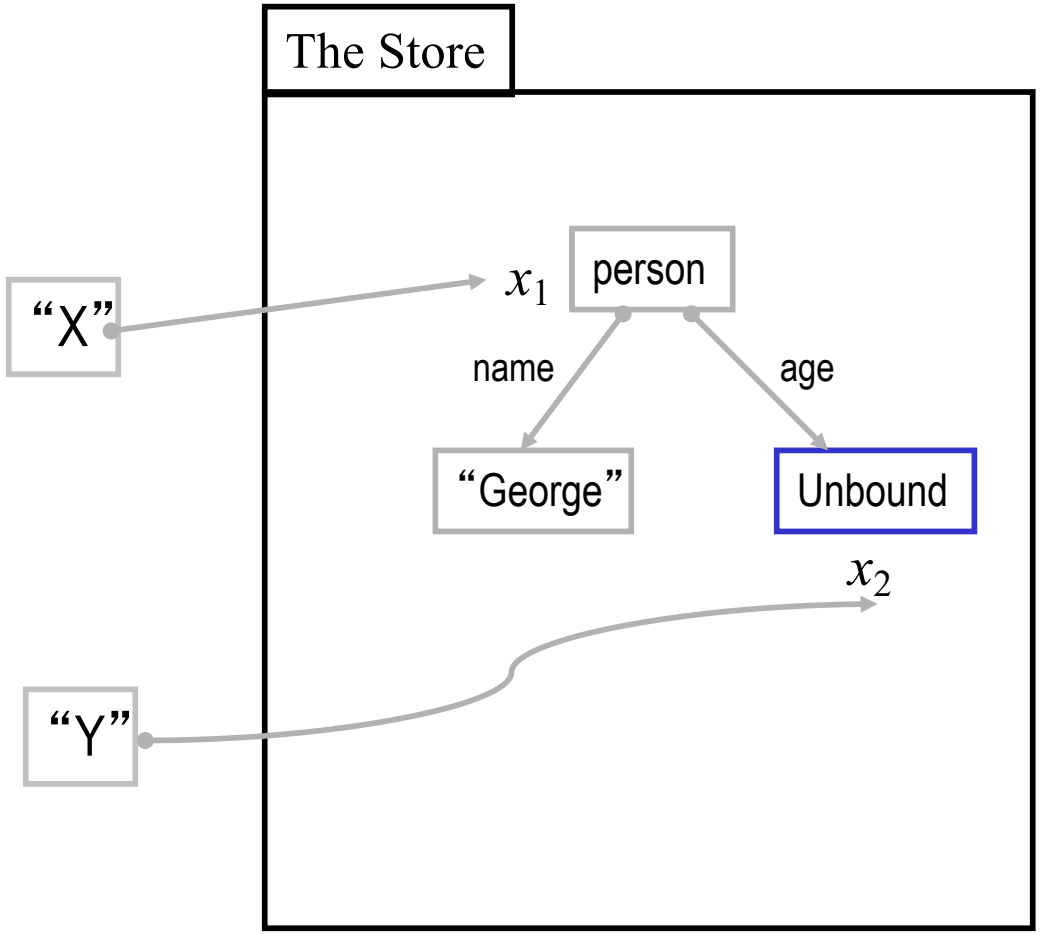
```
fun {Rectangle}
   W H in W::0#9   H::0#9
   W+H =: 10
   W*H >=: 24
   W <: H
   {FD.distribute naive rect(W H)}
   rect(W H)
end
{Browse {Search.base.all Rectangle}}
% displays [rect(4 6)]
```

Constraint-based Computation Model

- Constraints are of two kinds:
  - *Basic constraints*: represented directly in the single-assignment store. For example, X in {0..9}.
  - *Propagators*: constraints represented as threads that use local deduction to propagate information across partial values by adding new basic constraints. For example, X+Y =: 10.
- A *computation space* encapsulates basic constraints and propagators. Spaces can be nested, to support distribution and search strategies.
  - Distribution strategies determine how to create new computation spaces, e.g., a subspace assuming X = 4 and another with X \= 4.
  - Search strategies determine in which order to consider subspaces, e.g., depth-first search or breadth-first search.

Partial Values (Review)

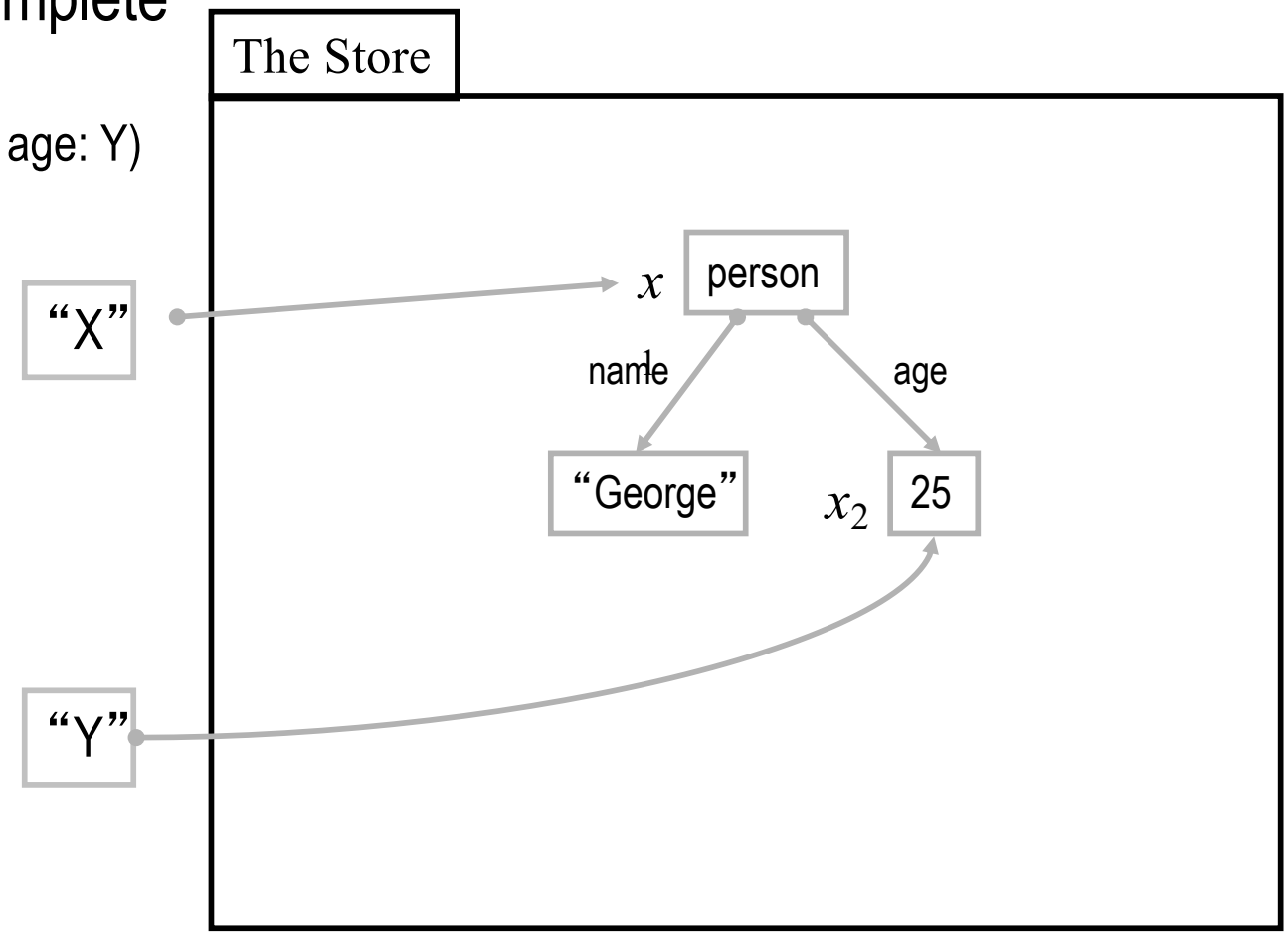
- Is a data structure that may contain unbound variables
- The store contains the partial value: person(name: “George” age: x_2)
- declare Y X
  X = person(name: “George” age: Y)
- The identifier ’Y’ refers to x_2



Partial Values (2)

Partial Values may be complete

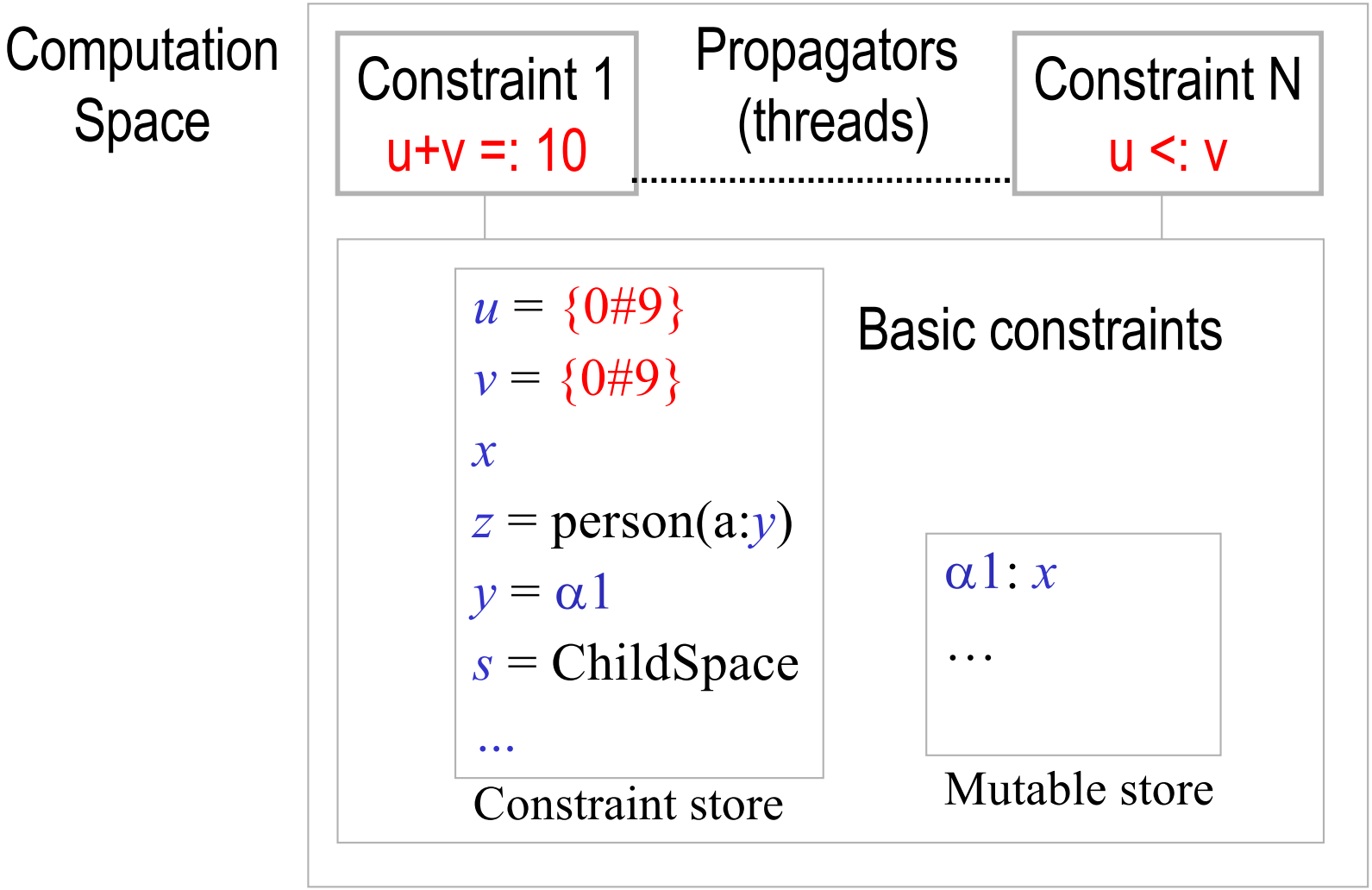
- declare Y X
  X = person(name: “George” age: Y)
- **Y = 25**



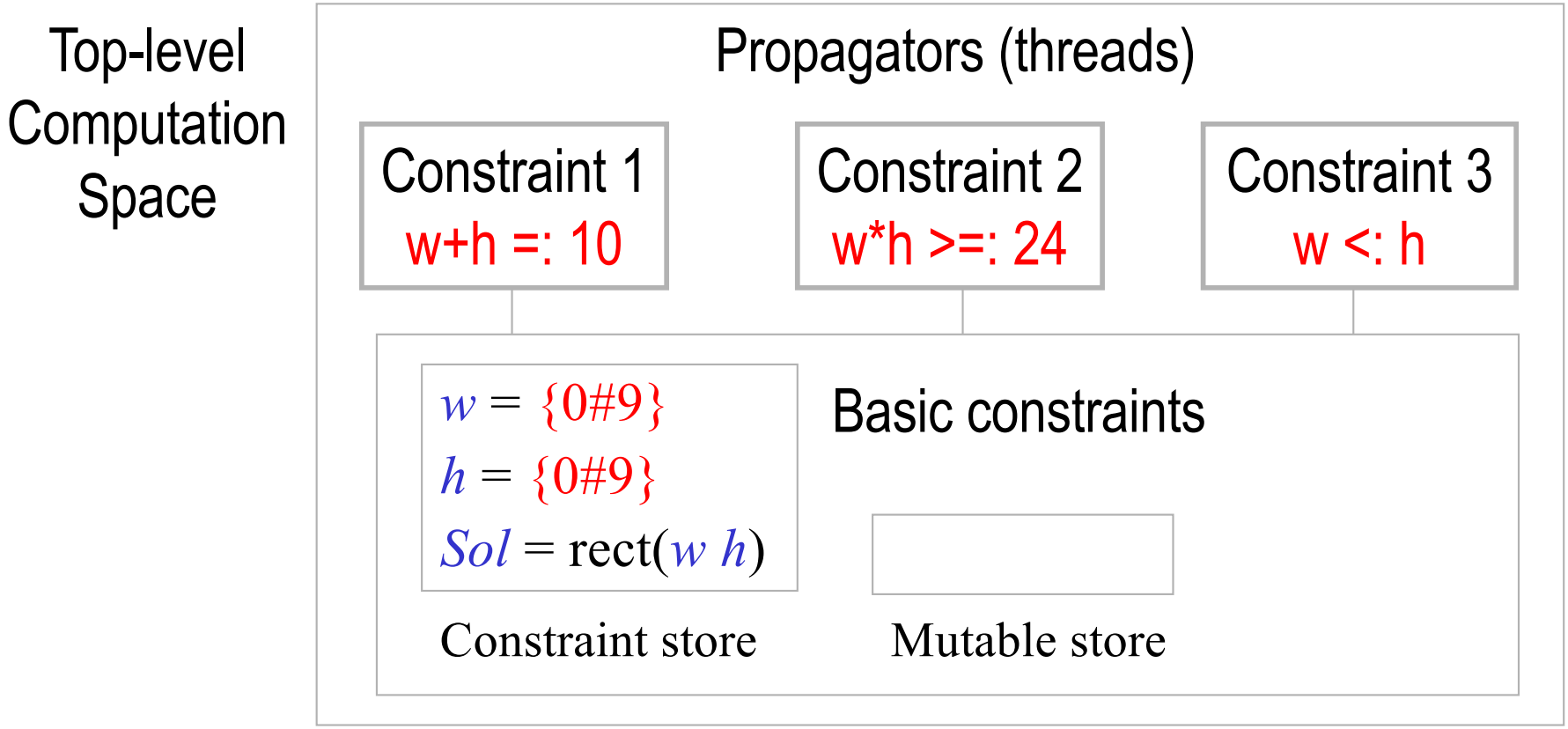
Variables and partial values

- Declarative variable:
  - is an entity that resides in a single-assignment store, that is initially unbound, and can be bound to exactly one (partial) value
  - it can be bound to several (partial) values as long as they are compatible with each other
- Partial value:
  - is a data-structure that may contain unbound variables
  - When one of the variables is bound, it is replaced by the (partial) value it is bound to
  - A complete value, or value for short is a data-structure that does not contain any unbound variables

Constraint-based computation model

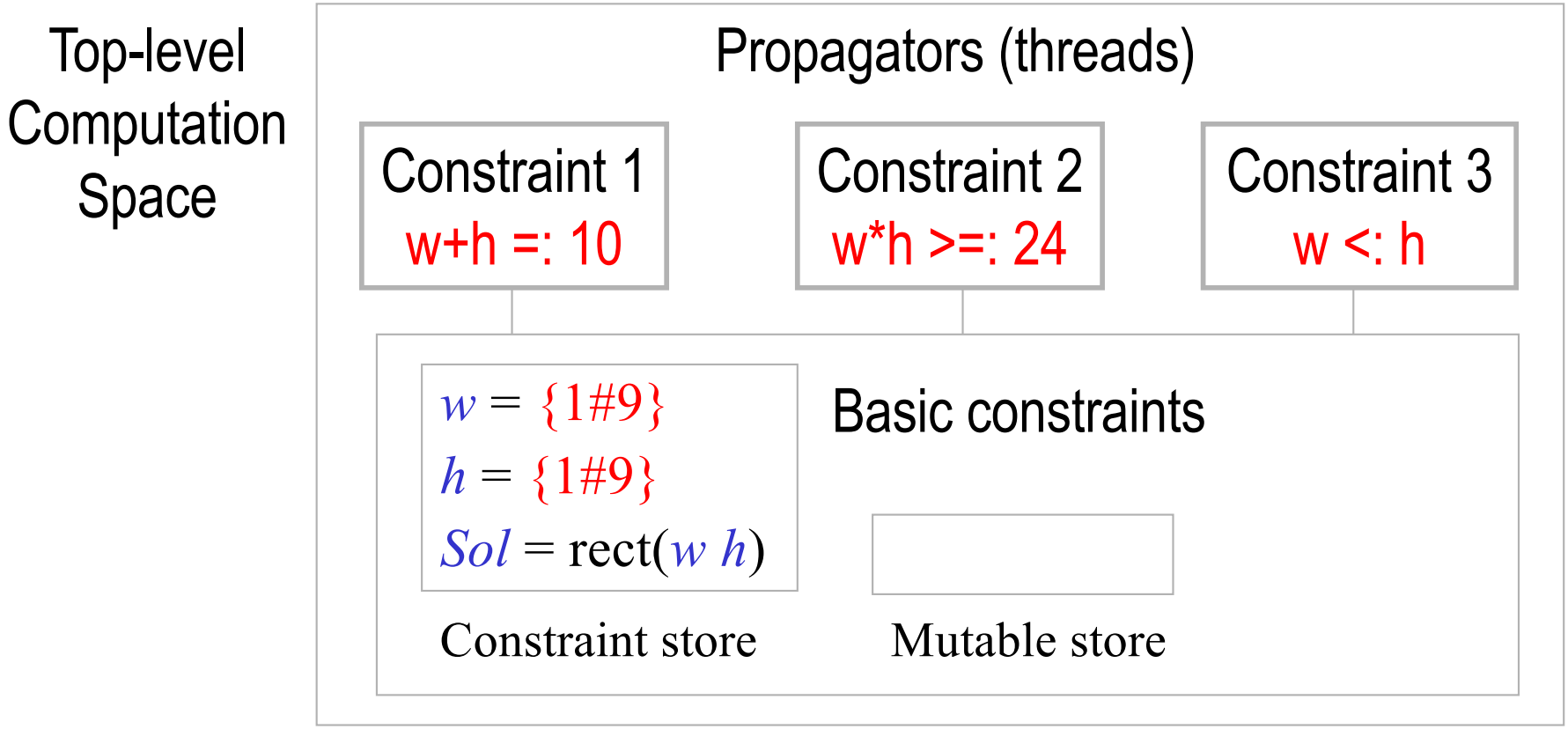


Constraint propagation: Rectangle example



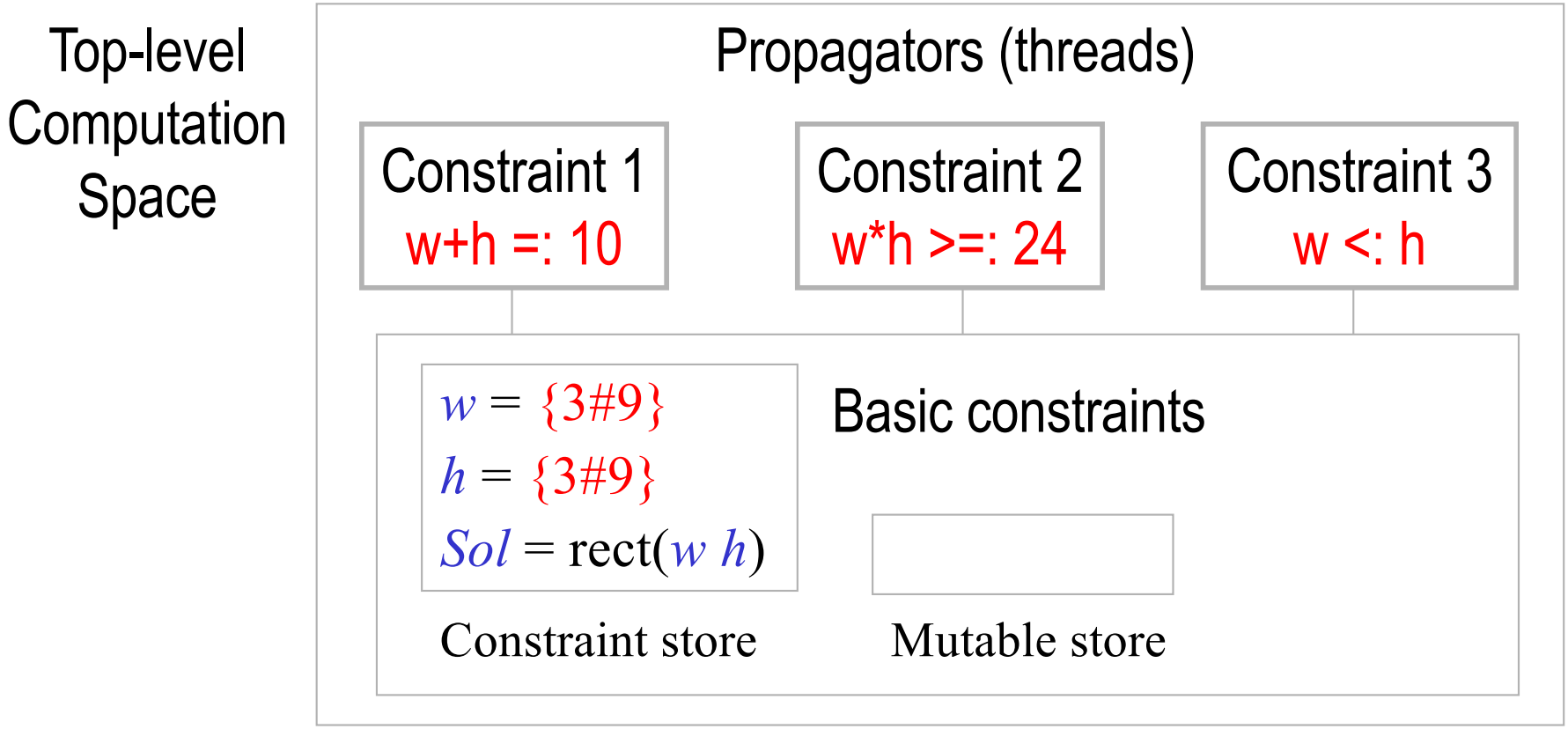
Let us consider propagator 1: w+h =: 10 → w cannot be 0; h cannot be 0.

Constraint propagation: Rectangle example



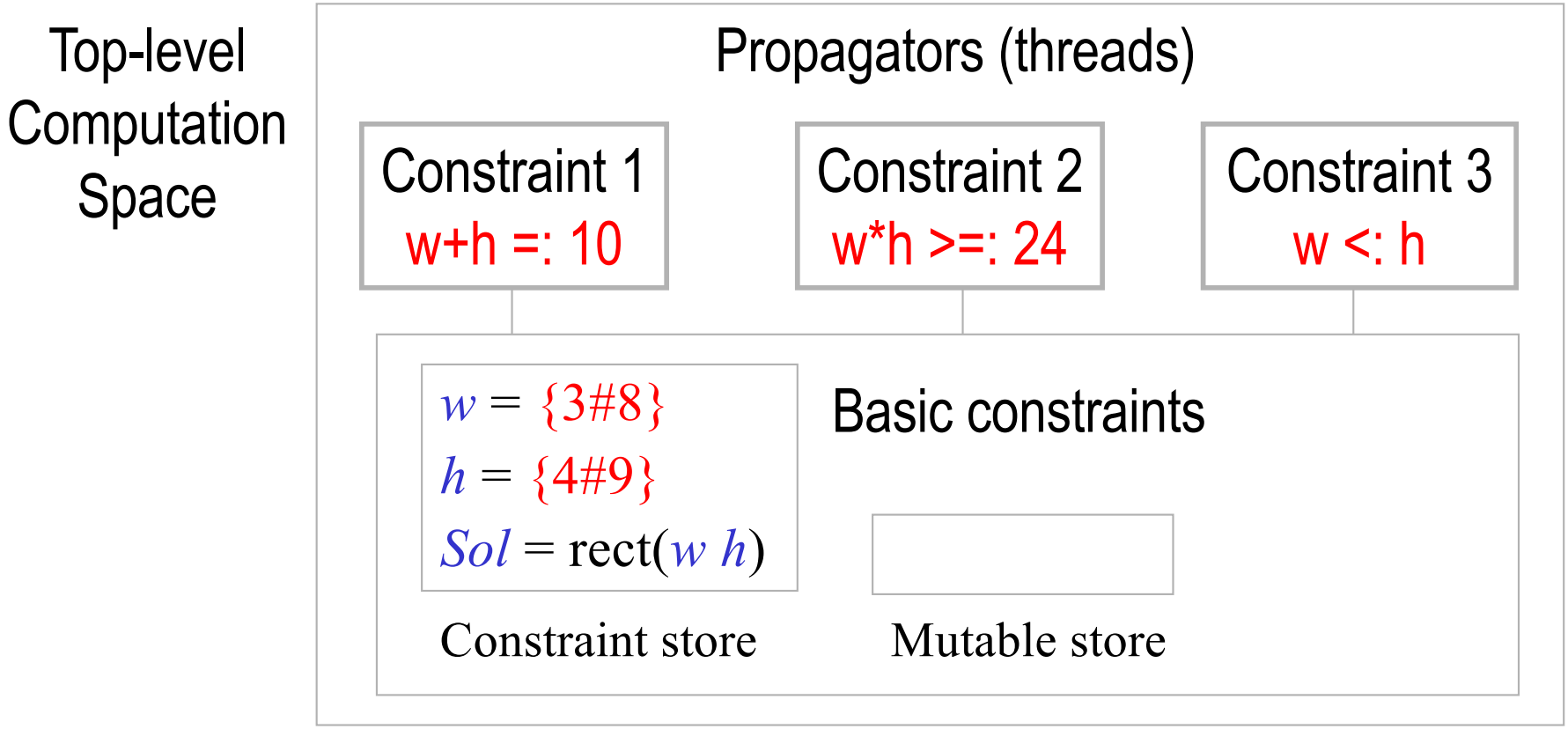
Let us consider propagator 2: w*h >=: 24 → w or h cannot be 1 or 2.

Constraint propagation: Rectangle example



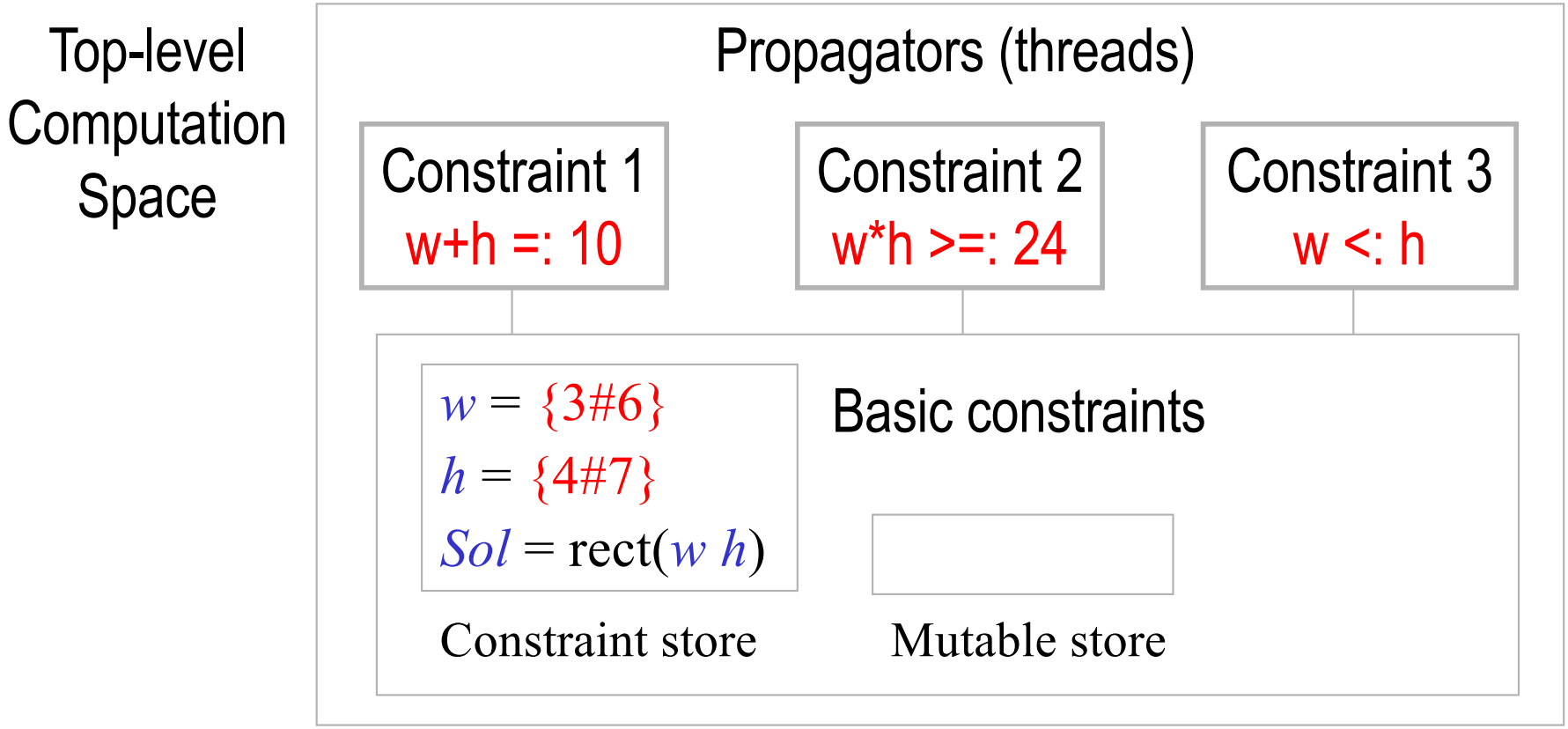
Let us consider propagator 3: w <: h → w cannot be 9, h cannot be 3.

Constraint propagation: Rectangle example



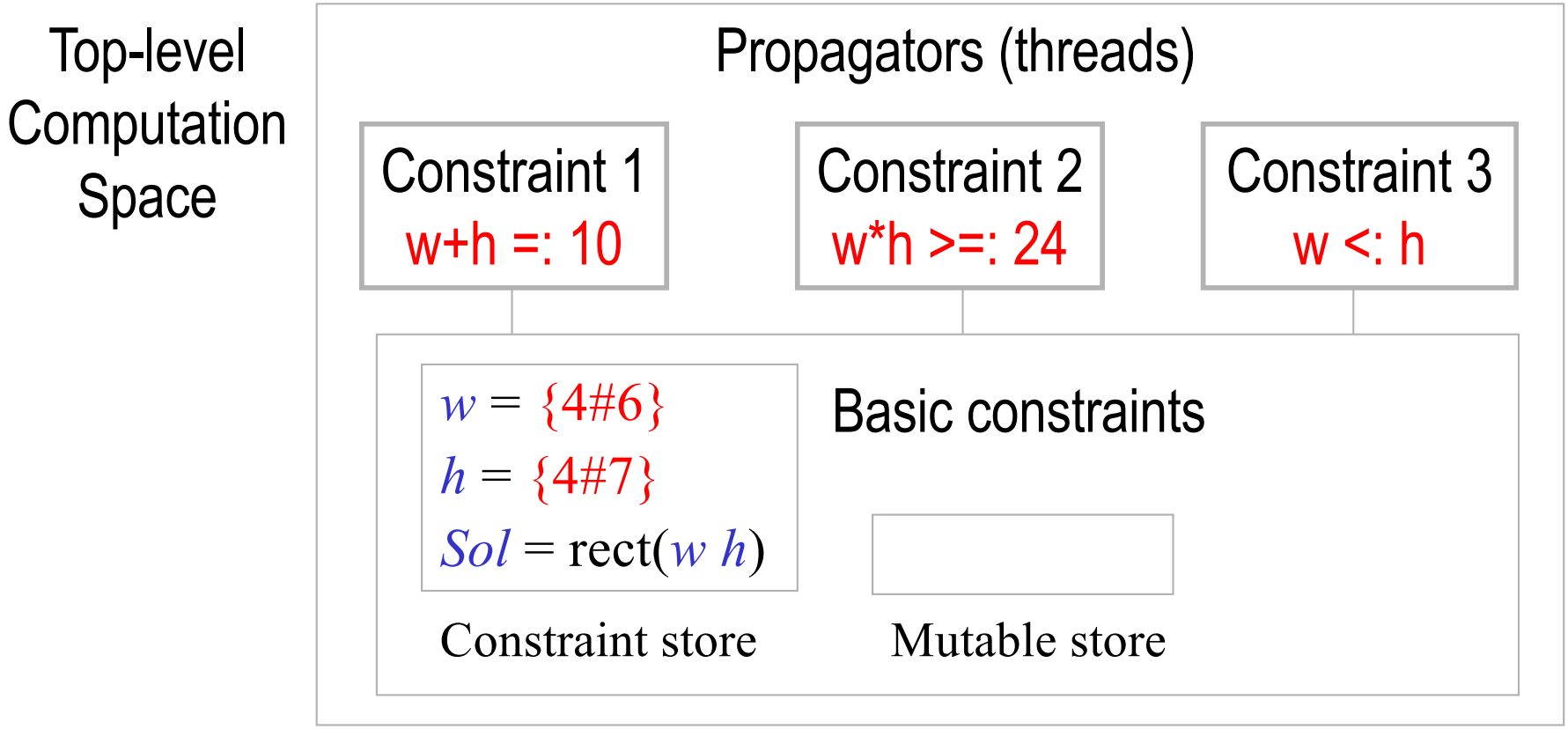
Let us consider propagator 1 **again**: w + h =: 10 →
w >= 3 implies h cannot be 8 or 9,
h >= 4 implies w cannot be 7 or 8.

Constraint propagation: Rectangle example



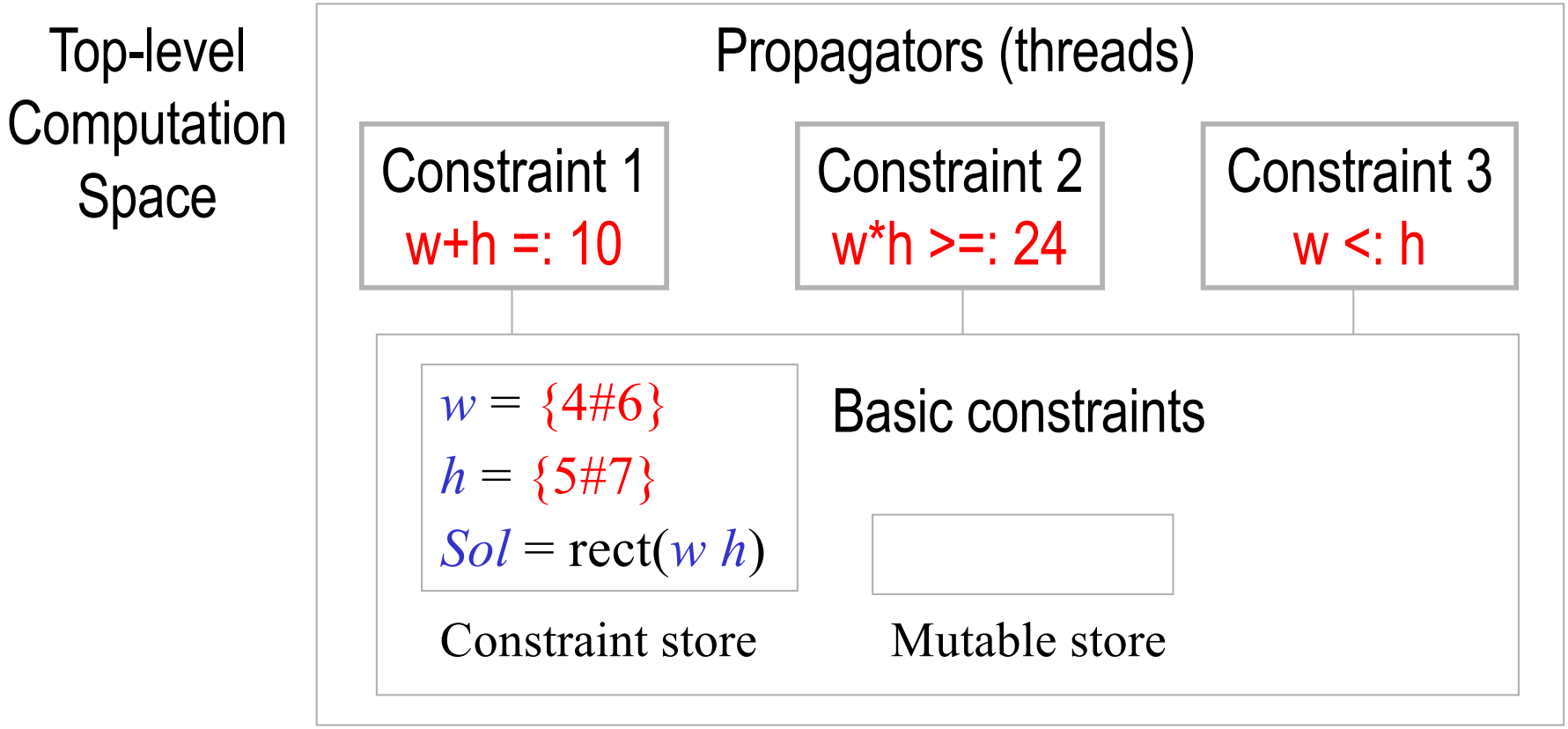
Let us consider propagator 2 **again**: w * h >=: 24 → h <= 7 implies w cannot be 3.

Constraint propagation: Rectangle example



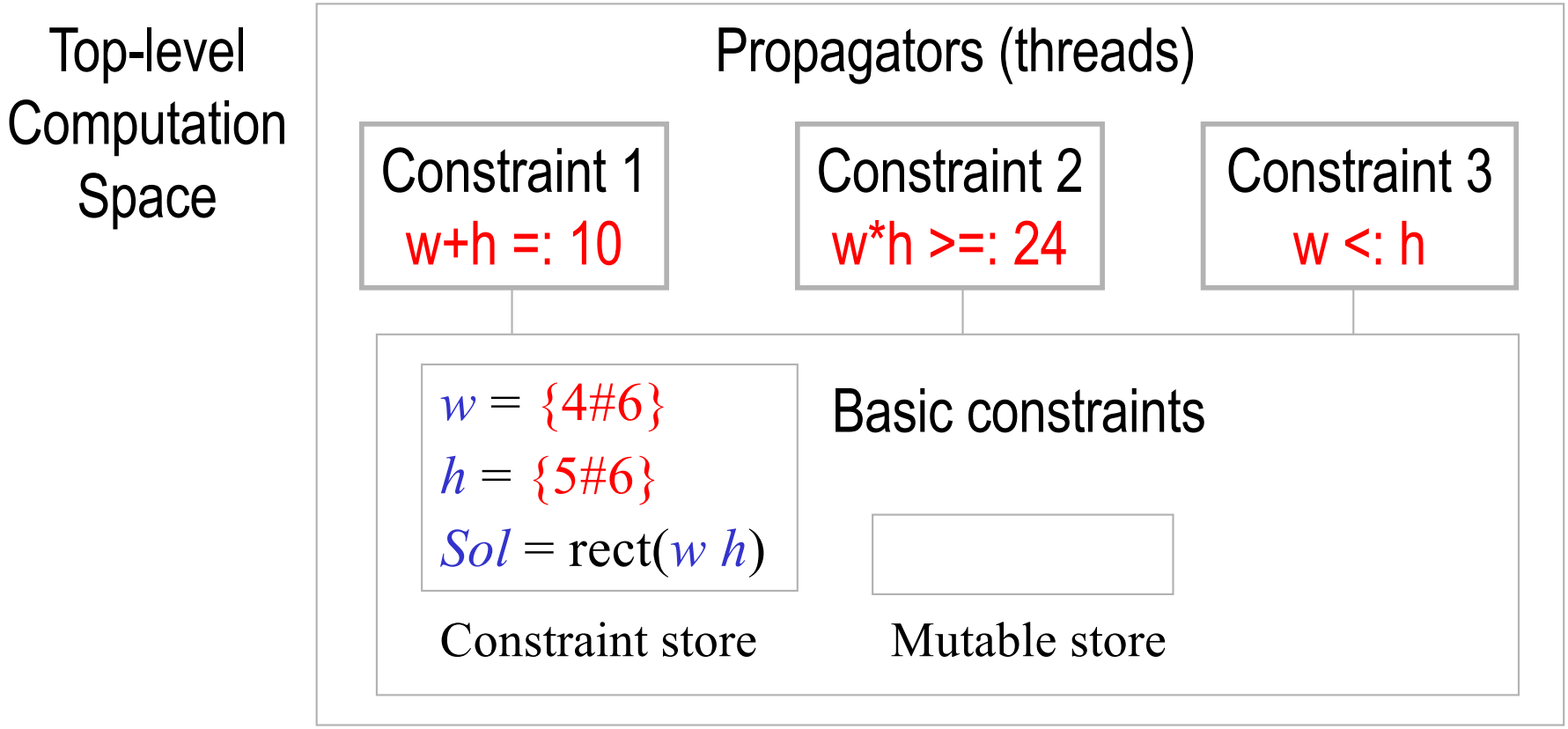
Let us consider propagator 3 **again**: w <: h → h cannot be 4.

Constraint propagation: Rectangle example



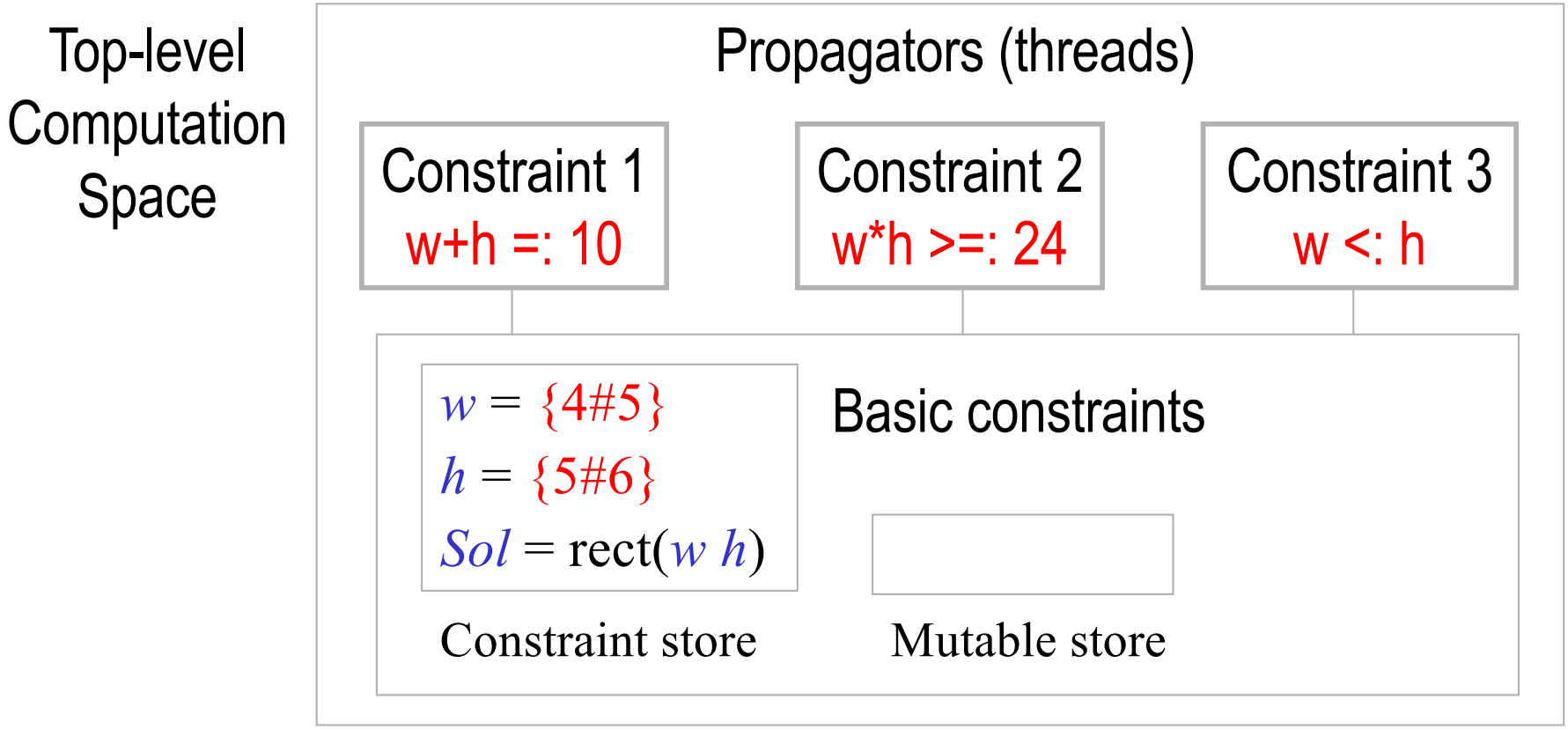
Let us consider propagator 1 **once more**: w + h =: 10 → h cannot be 7.

Constraint propagation: Rectangle example



Let us consider propagator 3 **once more**: w <: h → w cannot be 6.

Constraint propagation: Rectangle example



We have reached a **stable** computation space state: no single propagator can add more information to the constraint store.

Search

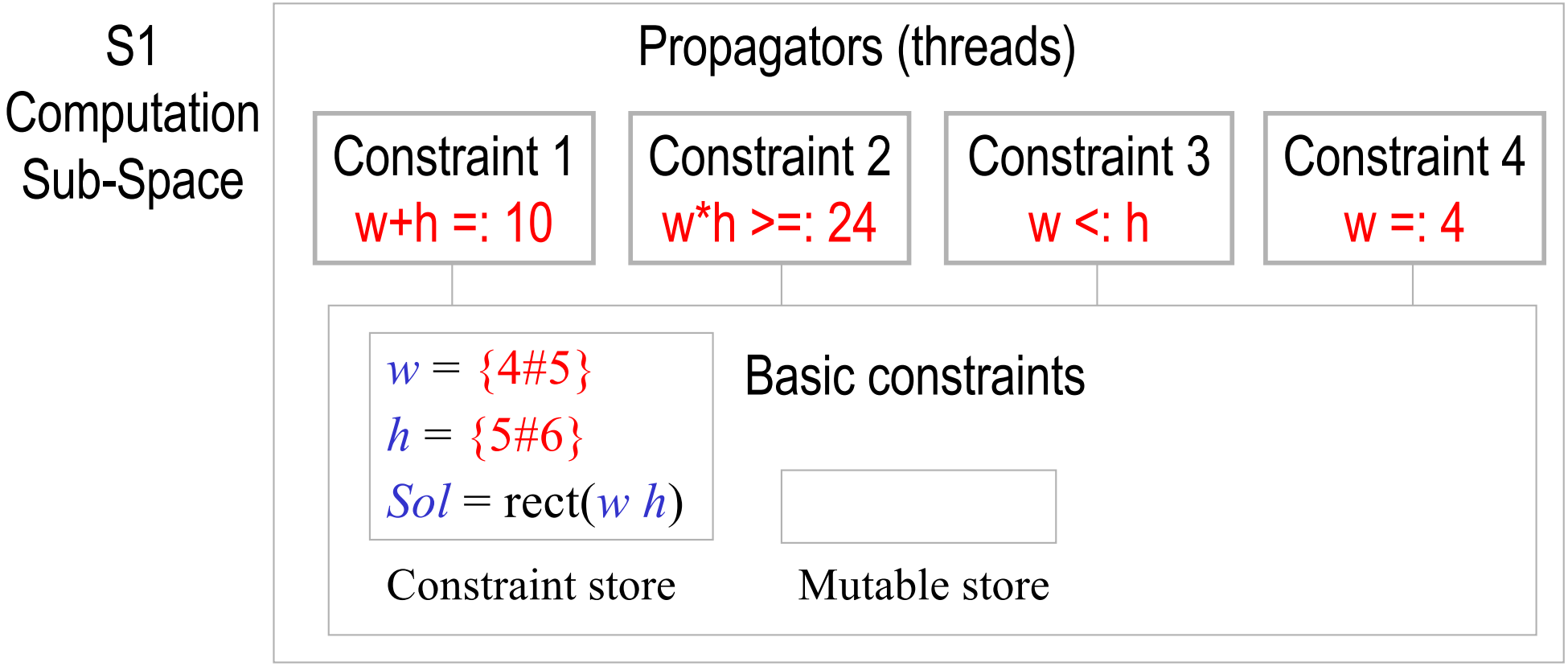
Once we reach a stable computation space (no local deductions can be made by individual propagators), we need to do search to make progress.

Divide original problem P into two new problems: (P ^ C) and (P ^ ¬C) and where C is a new constraint. The solution to P is the union of the solutions to the two new sub-problems.

In our Rectangle example, we divide the computation space S into two new sub-spaces S1 and S2 with new (respective) constraints:

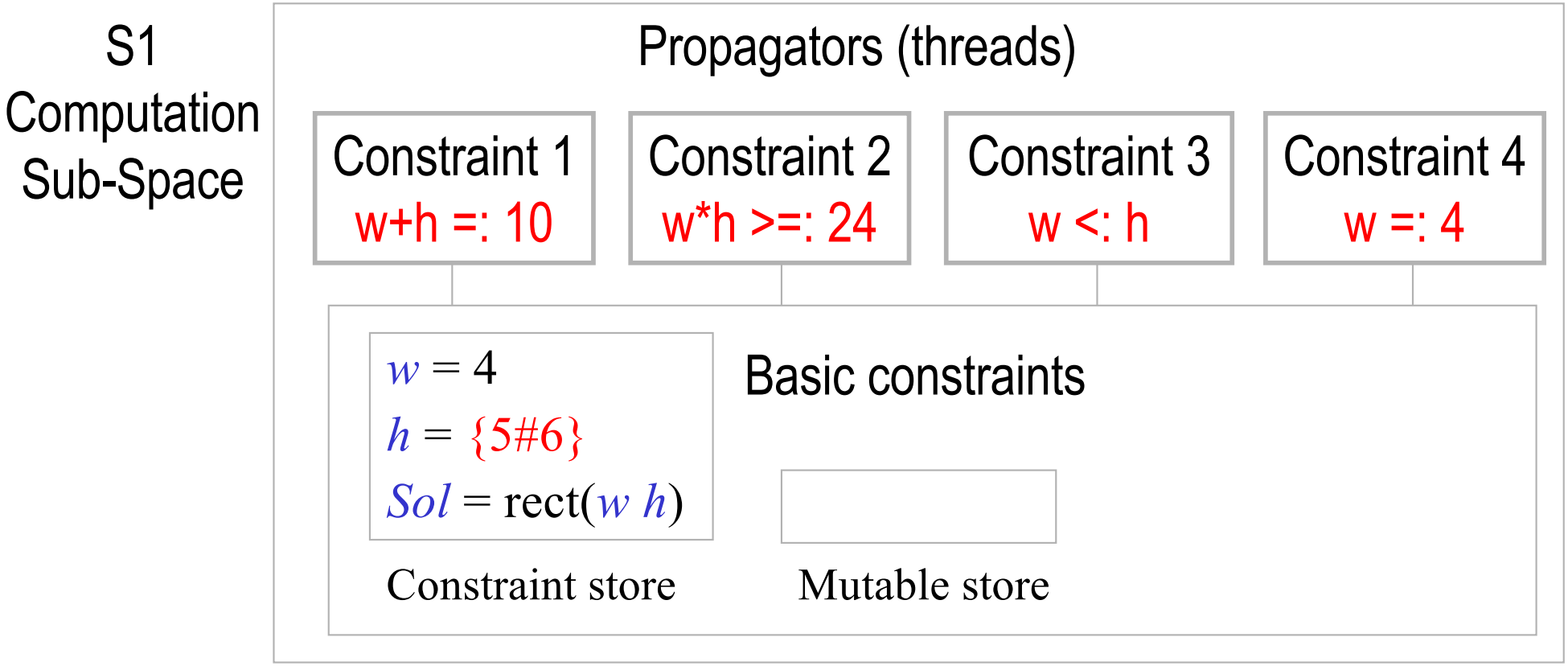
```
w =: 4
w \=: 4
```

Computation Space Search: Rectangle example with w=4



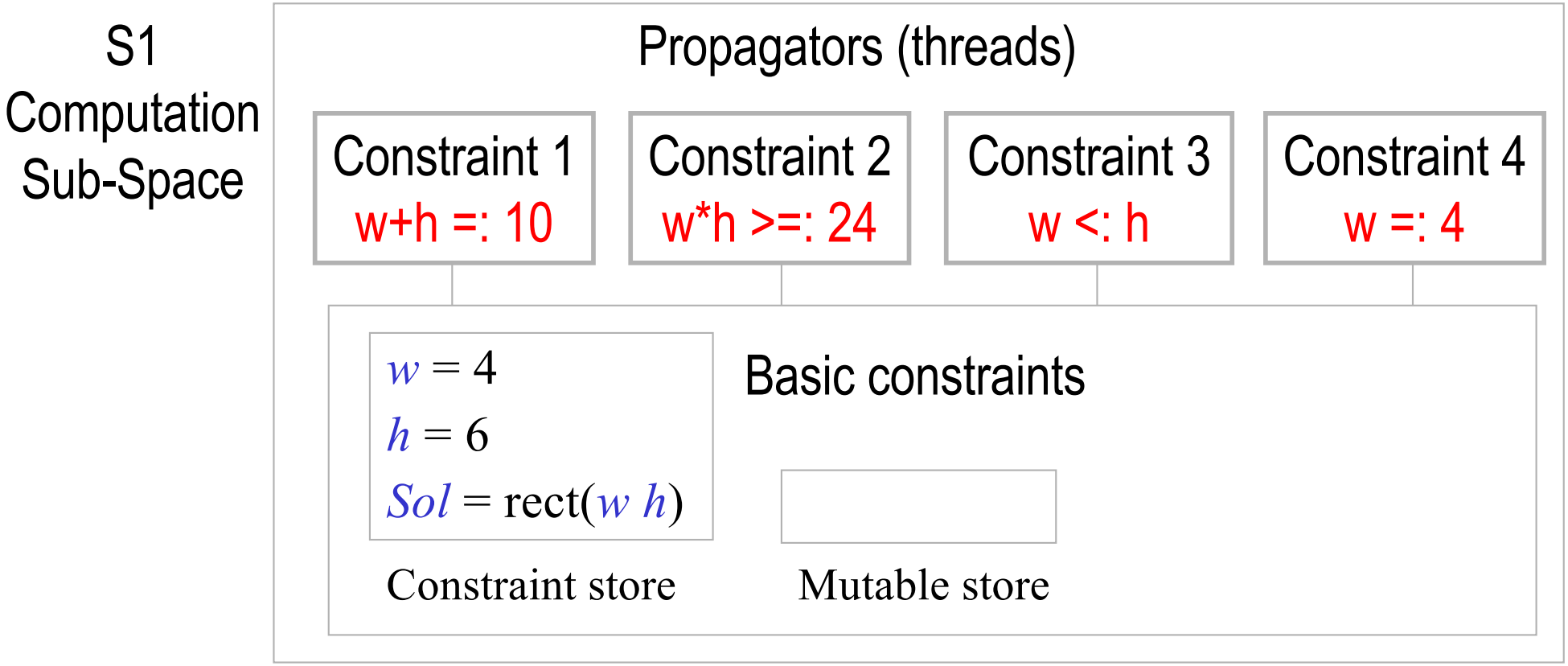
Constraint 4 implies that w = 4.

Computation Space Search: Rectangle example with w=4



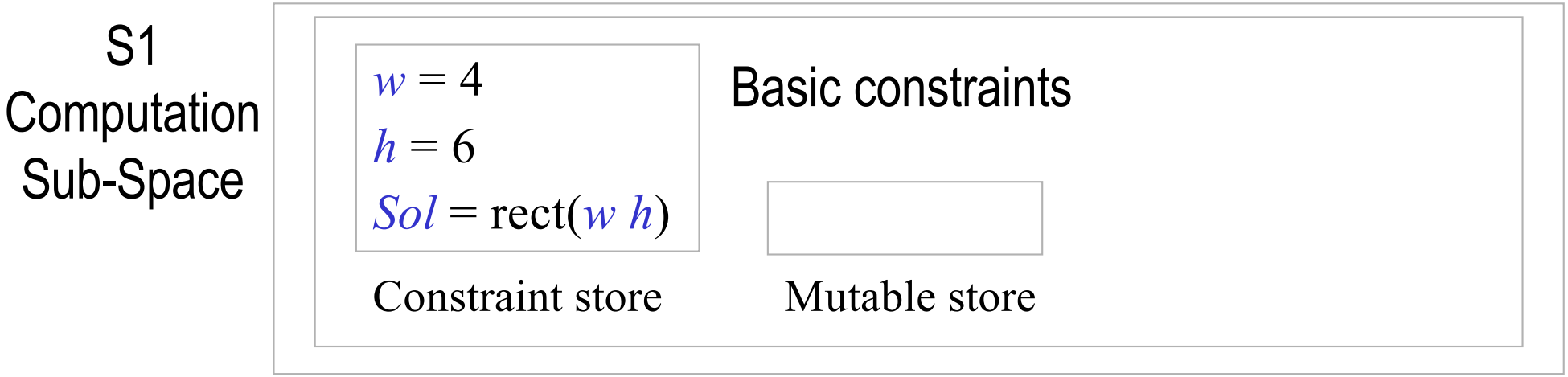
Constraint 1 or 2 implies that h = 6.

Computation Space Search: Rectangle example with w=4



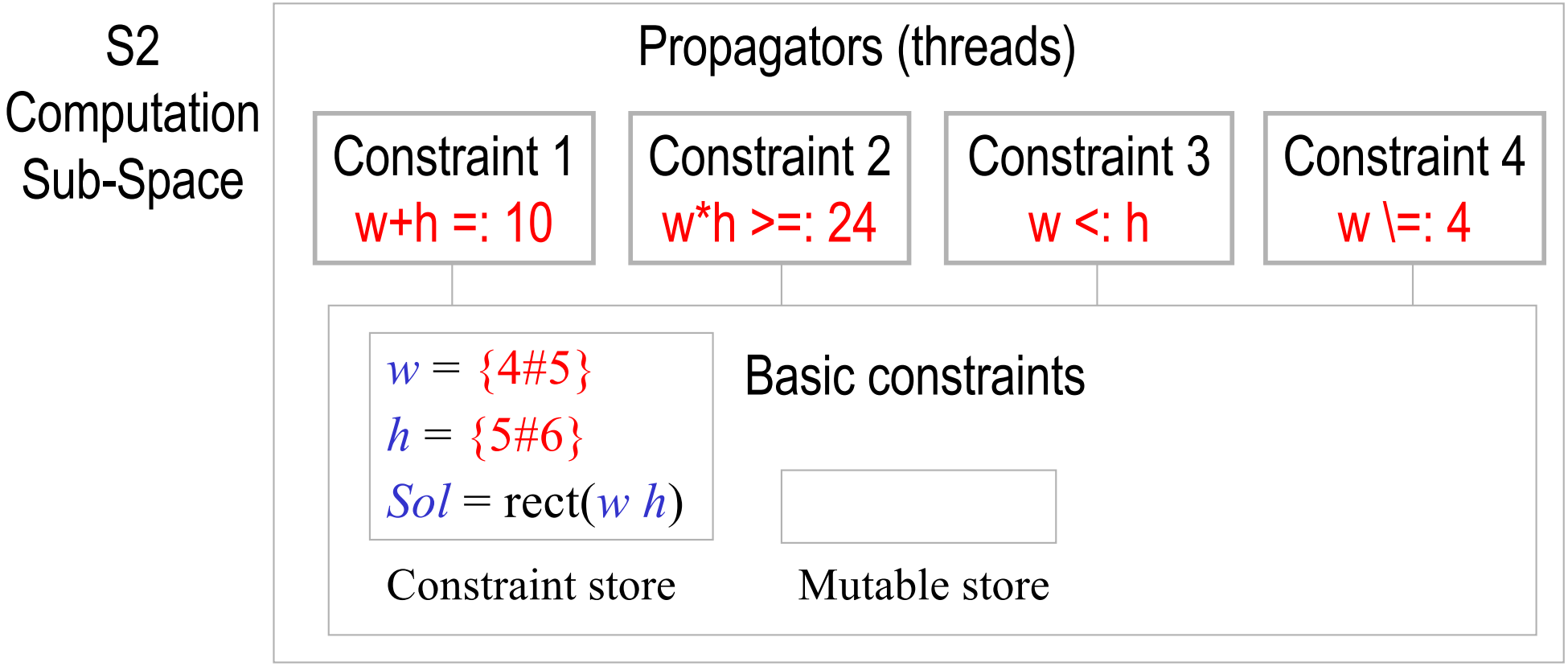
Since all the propagators are entailed by the store, their threads can terminate.

Computation Space Search: Rectangle example with w=4



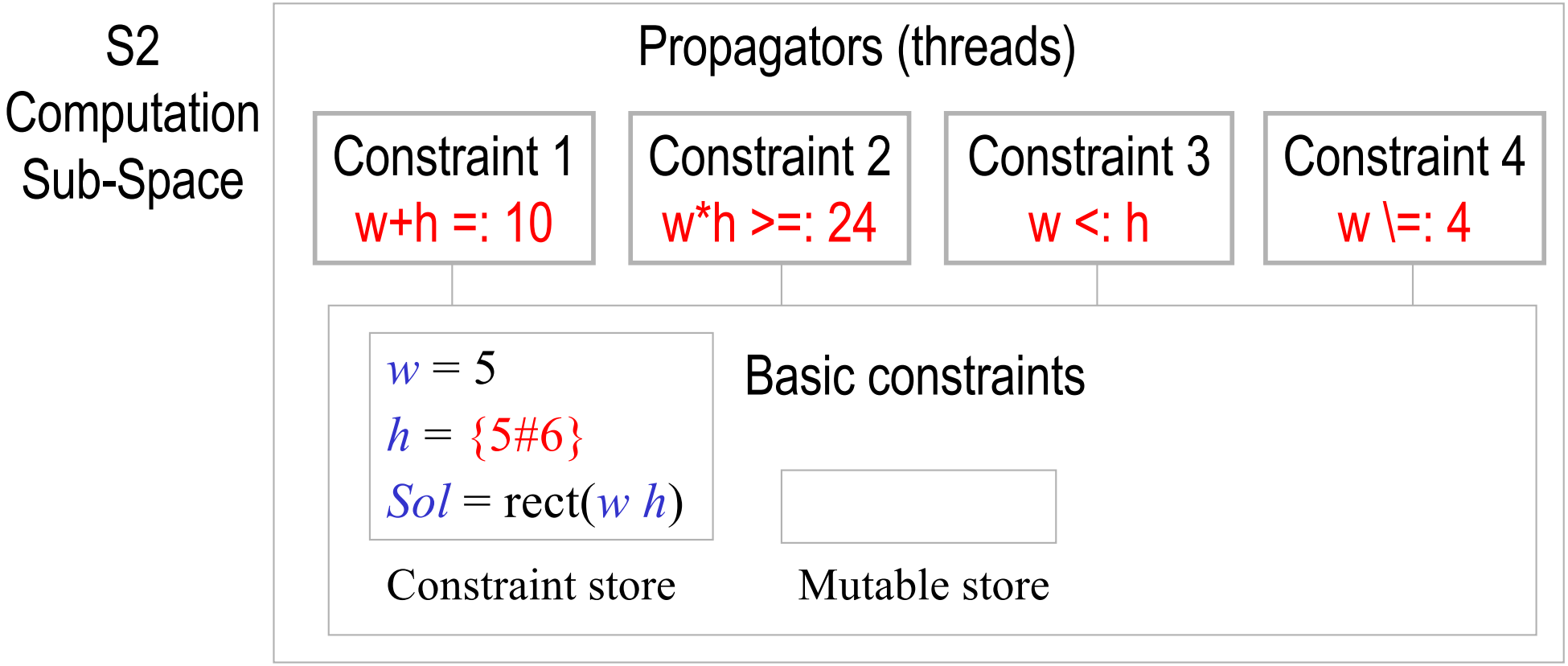
This is the final value store. A solution has been found.
The sub-space can now be **merged** with its parent computation space.

Computation Space Search: Rectangle example with w\=4



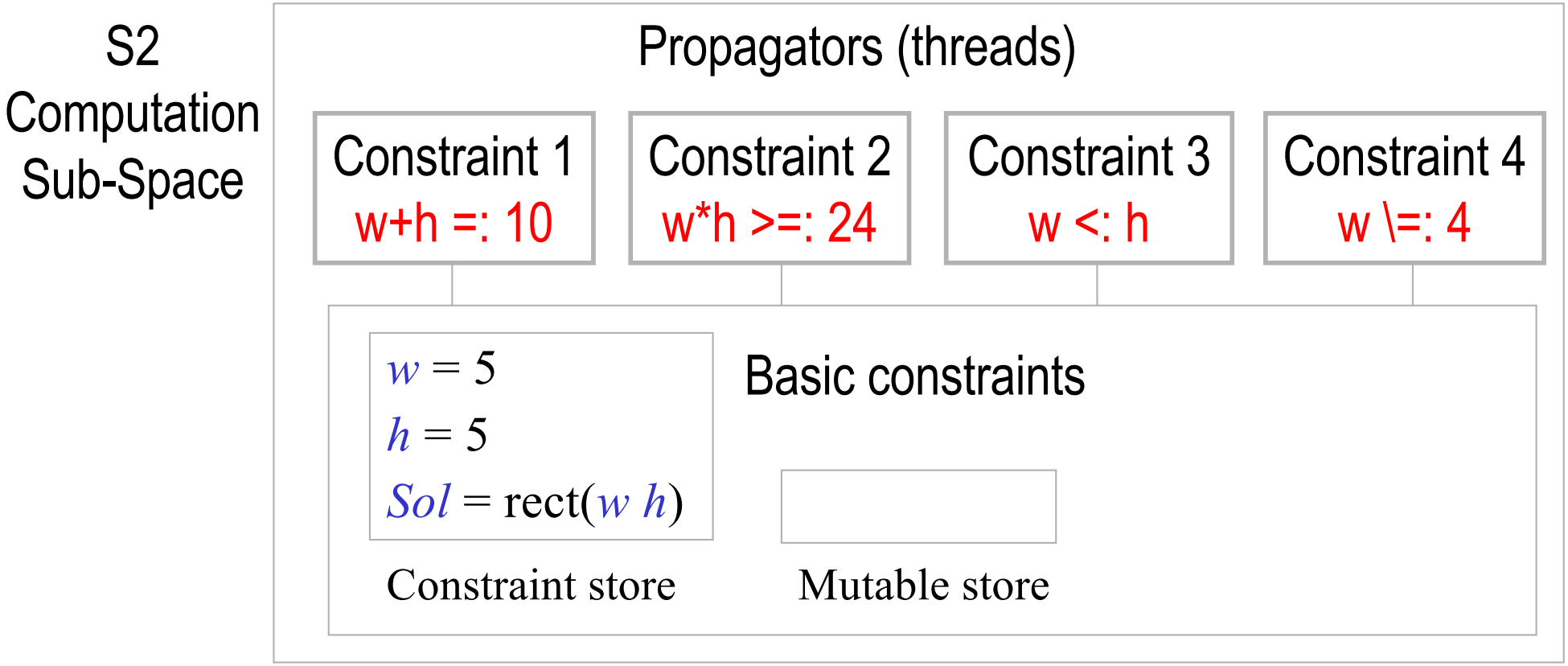
Constraint 4 implies that w = 5.

Computation Space Search: Rectangle example with w\=4



Constraint 1, w+h =: 10 → h = 5.

Computation Space Search: Rectangle example with w\=4



Constraint 3, w <: h, cannot be satisfied: computation sub-space S2 **fails**.

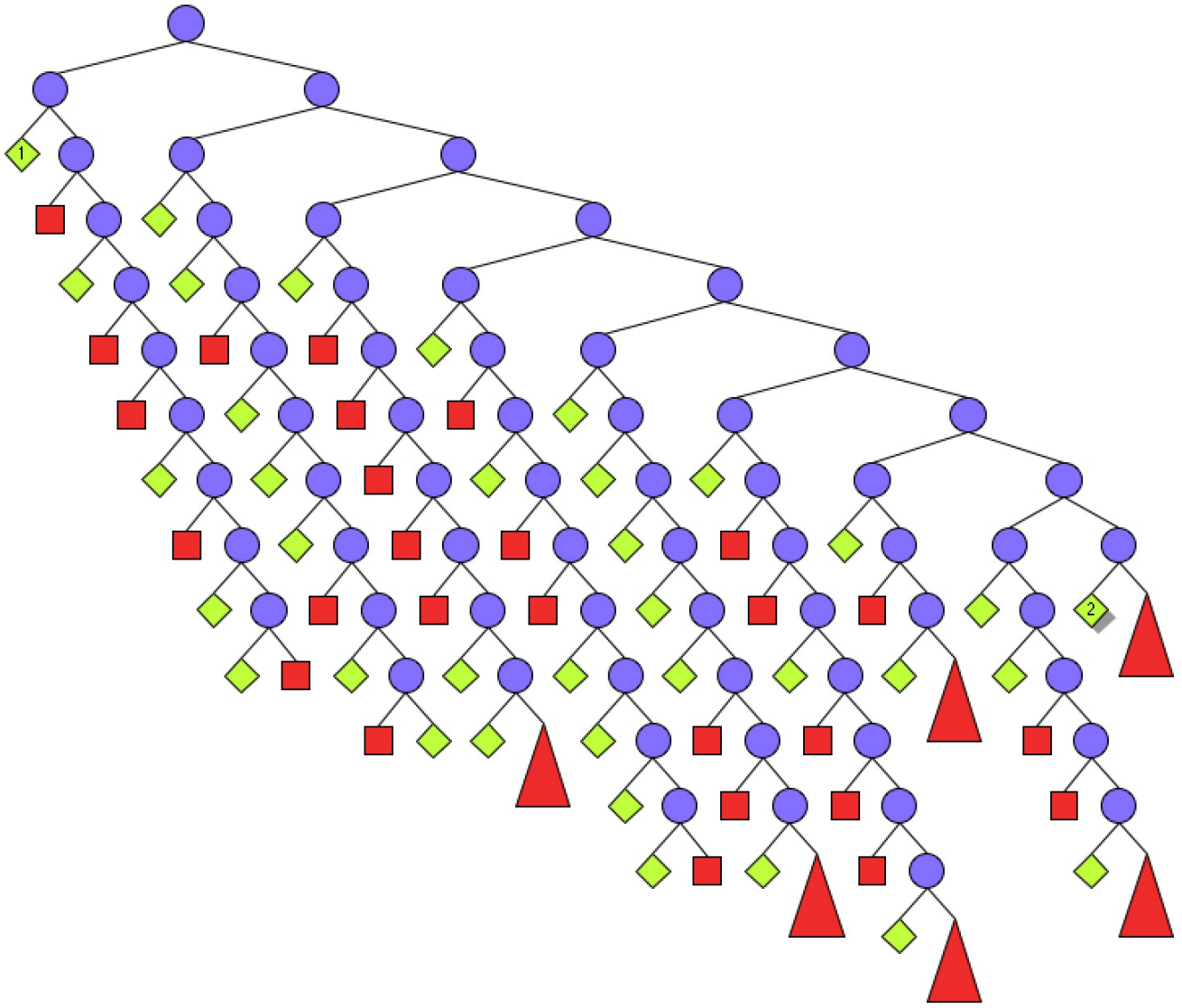
Finding palindromes (revisited)

- Find all four-digit palindromes that are products of two-digit numbers:

```
fun {Palindrome}
   A B C X Y in
   A::1000#9999   B::0#99 C::0#99
   A =: B*C
   X::1#9   Y::0#9
   A =: X*1000+Y*100+Y*10+X
   {FD.distribute ff [X Y]}
   A
end
{Browse {Search.base.all Palindrome}}          % 36 solutions
```

Computation spaces for Palindrome with Explorer

- At top-level, we have X=1, X\=1.
- Green diamonds correspond to **successful** sub-spaces.
- Red squares correspond to **failed** sub-spaces.



Programming Search with Computation Spaces

- The *search strategy* specifies the order to consider nodes in the search tree, e.g., depth-first search.
- The *distribution strategy* specifies the shape and content of the tree, i.e., how many alternatives exist at a node and what constraints are added for each alternative.
- They can be independent of each other. Distribution strategy is decided within the computation space. Search strategy is decided outside the computation space.

Programming Search with Computation Spaces

- Create the space with program (variables and constraints).
- Program runs in space: variables and propagators are created. Space executes until it reaches stability.
- Computation can create a choice point. Distribution strategy decides what constraint to add for each alternative. Computation inside space is suspended.
- Outside the space, if no choice point has been created, execution stops and returns a solution. Otherwise, search strategy decides what alternative to consider next and commits to that.

Primitive Operations for Computation Spaces

⟨statement⟩	::=	{NewSpace ⟨x⟩ ⟨y⟩}
	|	{WaitStable}
	|	{Choose ⟨x⟩ ⟨y⟩}
	|	{Ask ⟨x⟩ ⟨y⟩}
	|	{Commit ⟨x⟩ ⟨y⟩}
	|	{Clone ⟨x⟩ ⟨y⟩}
	|	{Inject ⟨x⟩ ⟨y⟩}
	|	{Merge ⟨x⟩ ⟨y⟩}

Depth-first single-solution search

```
fun {DFE S}
   case {Ask S}
   of failed then nil
   [] succeeded then [S]
   [] alternatives(2) then C={Clone S} in
      {Commit S 1}
      case {DFE S} of nil then {Commit C 2} {DFE C}
      [] [T] then [T]
      end
   end
end
```

```
% Given {Script Sol}, returns
solution [Sol] or nil:
fun {DFS Script}
   case {DFE {NewSpace Script}}
   of nil then nil
   [] [S] then [{Merge S}]
   end
end
```

Relational computation model (Oz)

The following defines the syntax of a statement, ⟨s⟩ denotes a statement

⟨s⟩	::= skip	*empty statement*
	| ⟨x⟩ = ⟨y⟩	*variable-variable binding*
	| ⟨x⟩ = ⟨v⟩	*variable-value binding*
	| ⟨s_1⟩ ⟨s_2⟩	*sequential composition*
	| local ⟨x⟩ in ⟨s_1⟩ end	*declaration*
	| proc {⟨x⟩ ⟨y_1⟩ … ⟨y_n⟩ } ⟨s_1⟩ end	*procedure introduction*
	| if ⟨x⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*conditional*
	| { ⟨x⟩ ⟨y_1⟩ … ⟨y_n⟩ }	*procedure application*
	| case ⟨x⟩ of ⟨pattern⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*pattern matching*
	| **choice ⟨s_1⟩ [] … [] ⟨s_n⟩ end**	***choice***
	| **fail**	***failure***

Relational Computation Model

- Declarative model (purely functional) is extended with *relations*.
- The choice statement groups a set of alternatives.
  - Execution of choice statement chooses one alternative.
  - Semantics is to rollback and try other alternatives if a failure is subsequently encountered.
- The fail statement indicates that the current alternative is wrong.
  - A fail is implicit upon trying to bind incompatible values, e.g., 3=4. This is in contrast to raising an exception (as in the declarative model).

Search tree and procedure

- The search tree is produced by creating a new branch at each *choice point*.
- When fail is executed, execution « backs up » or backtracks to the most recent choice statement, which picks the next alternative (left to right).
- Each path in the tree can correspond to no solution (« fail »), or to a solution (« succeed »).
- A search procedure returns a lazy list of all solutions, ordered according to a depth-first search strategy.

Rainy/Snowy Example

```
fun {Rainy}
   choice
      seattle [] rochester
   end
end

fun {Cold}
   rochester
end

proc {Snowy X}
   {Rainy X}
   {Cold X}
end
```

```
% display all
{Browse
   {Search.base.all
         proc {$ C} {Rainy C} end}}
{Browse {Search.base.all Snowy}}

% new search engine
E = {New Search.object script(Rainy)}

% calculate and display one at a time
{Browse {E next($)}}
```

Implementing the Relational Computation Model

choice ⟨s_1⟩ [] … [] ⟨s_n⟩ **end**

is a linguistic abstraction translated to:

case {**Choose** **N**}
of **1** **then** ⟨s_1⟩
[] **2** **then** ⟨s_2⟩
…
[] **N** **then** ⟨s_n⟩
end

Implementing the Relational Computation Model

```
% Returns the list of solutions of Script given by a
      lazy depth-first exploration
fun {Solve Script}
   {SolveStep {Space.new Script} nil}
end

% Returns the list of solutions of S appended with
      SolTail
fun {SolveStep S SolTail}
   case {Space.ask S}
   of failed              then SolTail
   [] succeeded           then {Space.merge S}|SolTail
   [] alternatives(N) then {SolveLoop S 1 N SolTail}
   end
end
```

```
% Lazily explores the alternatives I through
      N of space S, and returns the list of
      solutions found, appended with SolTail
fun lazy {SolveLoop S I N SolTail}
   if I>N then
      SolTail
   elseif I==N then
      {Space.commit S I}
      {SolveStep S SolTail}
   else
      C={Space.clone S}
      NewTail={SolveLoop S I+1 N SolTail}
   in
      {Space.commit C I}
      {SolveStep C NewTail}
   end
end
```

Exercises

97. Try different orders of execution for propagator threads. Do they always end up in the same constraint store? Why or why not?
98. CTM Exercise 12.6.1 (pg 774).
99. CTM Exercise 12.6.3 (pg 775).