Programming Languages (CSCI 4430/6430)

Part 3: Logic Programming: Summary

Carlos Varela
Rensselaer Polytechnic Institute

December 8, 2017

An Early (1971) "Conversation"

USER:

Cats kill mice.

Tom is a cat who does not like mice who eat cheese.

Jerry is a mouse who eats cheese.

Max is not a mouse.

What does Tom do?

COMPUTER:

Tom does not like mice who eat cheese.

Tom kills mice.

USER:

Who is a cat?

COMPUTER:

Tom.

USER:

What does Jerry eat?

COMPUTER:

Cheese.

USER:

Who does not like mice who eat cheese?

COMPUTER:

Tom.

USER:

What does Tom eat?

COMPUTER:

What cats who do not like mice who eat cheese eat.

Logic programming

- A program is a collection of *axioms*, from which theorems can be proven.
- A *goal* states the theorem to be proved.
- A logic programming language implementation attempts to satisfy the goal given the axioms and built-in inference mechanism.

First Order Predicate Calculus

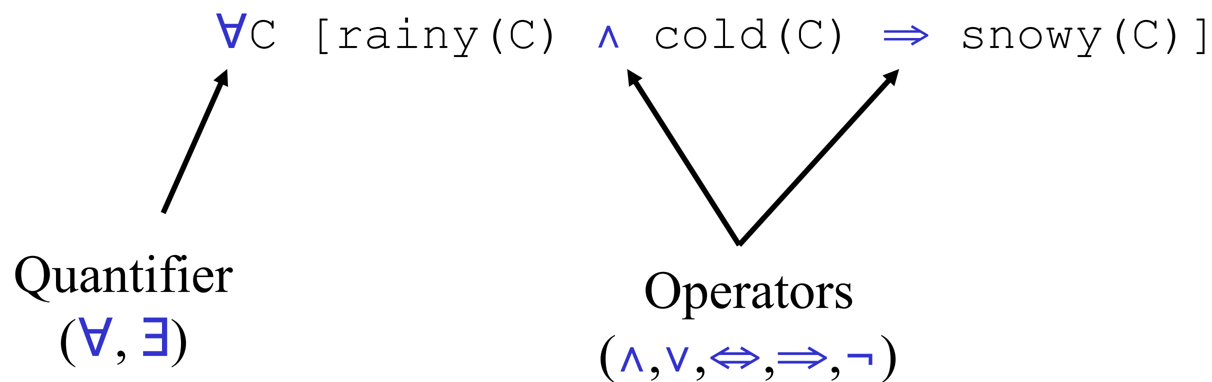
- Adds variables, terms, and (first-order) quantification of variables.
- Predicate syntax:

`a`	`::=`	$p(v_1, v_2, \ldots, v_n)$	predicate
`f`	`::=`	`a`	atom
	`|`	$v = p(v_1, v_2, \ldots, v_n)$	equality
	`|`	$v_1 = v_2$	
	`|`	$f \wedge f \mid f \vee f \mid f \Leftrightarrow f \mid f \Rightarrow f \mid \neg f$	
	`|`	$\forall v . f$	universal quantifier
	`|`	$\exists v . f$	existential quantifier

Predicate Calculus

- In mathematical logic, a *predicate* is a function that maps constants or variables to true and false.
- Predicate calculus enables reasoning about propositions.

- For example:



Structural Congruence Laws

$$P_1 \Rightarrow P_2 \equiv \neg P_1 \vee P_2$$

$$\neg \exists X\ [P(X)] \equiv \forall X\ [\neg P(X)]$$
$$\neg \forall X\ [P(X)] \equiv \exists X\ [\neg P(X)]$$

$$\neg (P_1 \wedge P_2) \equiv \neg P_1 \vee \neg P_2$$
$$\neg (P_1 \vee P_2) \equiv \neg P_1 \wedge \neg P_2$$
$$\neg \neg P \equiv P$$

$$(P_1 \Leftrightarrow P_2) \equiv (P_1 \Rightarrow P_2) \wedge (P_2 \Rightarrow P_1)$$

$$P_1 \vee (P_2 \wedge P_3) \equiv (P_1 \vee P_2) \wedge (P_1 \vee P_3)$$
$$P_1 \wedge (P_2 \vee P_3) \equiv (P_1 \wedge P_2) \vee (P_1 \wedge P_3)$$

$$P_1 \vee P_2 \equiv P_2 \vee P_1$$

Clausal Form

- Looking for a *minimal kernel* appropriate for theorem proving.
- Propositions are transformed into normal form by using structural congruence relationship.
- One popular normal form candidate is *clausal form*.
- Clocksin and Melish (1994) introduce a 5-step procedure to convert first-order logic propositions into clausal form.

Clocksin and Melish Procedure

1. Eliminate implication (⇒) and equivalence (⇔).
2. Move negation (¬) inwards to individual terms.
3. *Skolemization*: eliminate existential quantifiers (∃).
4. Move universal quantifiers (∀) to top-level and make implicit , i.e., all variables are universally quantified.
5. Use distributive, associative and commutative rules of ∨, ∧, and ¬, to move into *conjuctive normal form*, i.e., a conjuction of disjunctions (or *clauses*.)

Example

```
∀A [¬student(A) ⇒ (¬dorm_resident(A) ∧
                   ¬∃B [takes(A,B) ∧ class(B)])]
```

1. Eliminate implication (⇒) and equivalence (⇔).

```
∀A [student(A) ∨ (¬dorm_resident(A) ∧
                 ¬∃B [takes(A,B) ∧ class(B)])]
```

2. Move negation (¬) inwards to individual terms.

```
∀A [student(A) ∨ (¬dorm_resident(A) ∧
                 ∀B [¬(takes(A,B) ∧ class(B))])]

∀A [student(A) ∨ (¬dorm_resident(A) ∧
                 ∀B [¬takes(A,B) ∨ ¬class(B)])]
```

Example Continued

```
∀A [student(A) ∨ (¬dorm_resident(A) ∧
                  ∀B [¬takes(A,B) ∨ ¬class(B)])]
```

3. *Skolemization*: eliminate existential quantifiers (∃).
4. Move universal quantifiers (∀) to top-level and make implicit , i.e., all variables are universally quantified.

```
student(A) ∨ (¬dorm_resident(A) ∧
             (¬takes(A,B) ∨ ¬class(B)))
```

5. Use distributive, associative and commutative rules of ∨, ∧, and ¬, to move into *conjuctive normal form*, i.e., a conjuction of disjunctions (or *clauses*.)

```
(student(A) ∨ ¬dorm_resident(A)) ∧
(student(A) ∨ ¬takes(A,B) ∨ ¬class(B))
```

Horn clauses

- A standard form for writing axioms, e.g.:

```
father(X,Y) ⇐ parent(X,Y), male(X).
```

- The Horn clause consists of:
  - A *head* or consequent term H, and
  - A *body* consisting of terms B_i

$$H \Leftarrow B_0 , B_1 , \ldots, B_n$$

- The semantics is:

« If B_0 , B_1 , …, and B_n, then H »

Clausal Form to Prolog

```
(student(A) ∨ ¬dorm_resident(A)) ∧
(student(A) ∨ ¬takes(A,B) ∨ ¬class(B))
```

6. Use commutativity of ∨ to move negated terms to the right of each clause.
7. Use $P_1 \lor \neg P_2 \equiv P_2 \Rightarrow P_1 \equiv P_1 \Leftarrow P_2$

```
(student(A) ⇐ dorm_resident(A)) ∧
(student(A) ⇐ ¬(¬takes(A,B) ∨ ¬class(B)))
```

8. Move Horn clauses to Prolog:

```
student(A) :- dorm_resident(A).
student(A) :- takes(A,B),class(B).
```

Skolemization

```
∃X [takes(X,cs101) ∧ class_year(X,2)]
```

introduce a Skolem constant to get rid of existential quantifier (∃):

```
takes(x,cs101) ∧ class_year(x,2)
```

```
∀X [¬dorm_resident(X) ∨
      ∃A [campus_address_of(X,A)]]
```

introduce a Skolem function to get rid of existential quantifier (∃):

```
∀X [¬dorm_resident(X) ∨
      campus_address_of(X,f(X))]
```

Limitations

- If more than one non-negated (positive) term in a clause, then it cannot be moved to a Horn clause (which restricts clauses to only one head term).
- If zero non-negated (positive) terms, the same problem arises (Prolog's inability to prove logical negations).
- For example:
  - « every living thing is an animal or a plant »

```
animal(X) ∨ plant(X) ∨ ¬living(X)
animal(X) ∨ plant(X) ⇐ living(X)
```

Prolog Terms

- Constants

```
rpi
troy
```

- Variables

```
University
City
```

- Predicates

```
located_at(rpi,troy)
pair(a, pair(b,c))
```

Can be nested.

Resolution

- To derive new statements, Robinson's resolution principle says that if two Horn clauses:

$$H_1 \Leftarrow B_{11}, B_{12}, \ldots, B_{1m}$$
$$H_2 \Leftarrow B_{21}, B_{22}, \ldots, B_{2n}$$

are such that $\mathbf{H_1}$ matches $\mathbf{B_{2i}}$, then we can replace $\mathbf{B_{2i}}$ with $\mathbf{B_{11}, B_{12}, \ldots, B_{1m}}$:

$$H_2 \Leftarrow B_{21}, B_{22}, \ldots, B_{2(i-1)}, \underline{B_{11}, B_{12}, \ldots, B_{1m}}, B_{2(i+1)} \ldots, B_{2n}$$

- For example:

$$\begin{array}{l} C \Leftarrow A,B \\ E \Leftarrow C,D \\ \hline E \Leftarrow A,B,D \end{array}$$

Resolution Example

```
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Y) :- father(X,Z), parent(Z,Y).
```

```
grandfather(X,Y) :-
          parent(X,Z), male(X), parent(Z,Y).
```

`:-` is Prolog's notation (syntax) for $\Leftarrow$.

Unification

- During *resolution*, free variables acquire values through *unification* with expressions in matching terms.
- For example:

```
male(carlos).
parent(carlos, tatiana).
parent(carlos, catalina).
father(X,Y) :- parent(X,Y), male(X).
____________________________________________
father(carlos, tatiana).
father(carlos, catalina).
```

Unification Process

- A constant unifies only with itself.
- Two predicates unify if and only if they have
  - the same *functor*,
  - the same number of *arguments*, and
  - the corresponding arguments *unify*.
- A variable unifies with anything.
  - If the other thing has a *value*, then the variable is *instantiated*.
  - If it is an *uninstantiated variable*, then the two variables are *associated*.

Backtracking

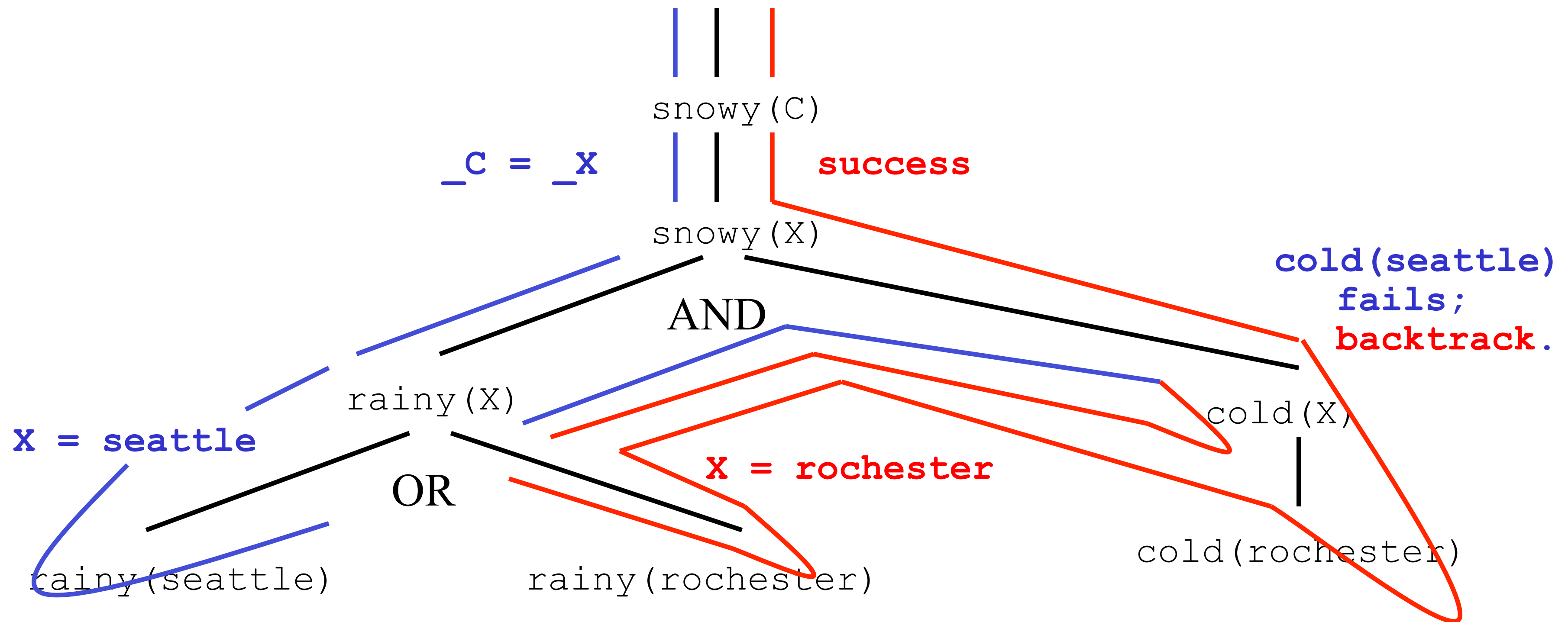
- *Forward chaining* goes from axioms forward into goals.
- *Backward chaining* starts from goals and works backwards to prove them with existing axioms.

Backtracking example

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```

Backtracking example

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```



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*
	|	$\langle s_1 \rangle$ $\langle s_2 \rangle$	*sequential composition*
	|	local ⟨x⟩ in $\langle s_1 \rangle$ end	*declaration*
	|	proc {⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ } $\langle s_1 \rangle$ end	*procedure introduction*
	|	if ⟨x⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*conditional*
	|	{ ⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ }	*procedure application*
	|	case ⟨x⟩ of ⟨pattern⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*pattern matching*
	|	**choice $\langle s_1 \rangle$ [] … [] $\langle s_n \rangle$ 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
```

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

{Browse {Search.base.all Snowy}}
```

Imperative Control Flow

- Programmer has *explicit control* on backtracking process.

`Cut (!)`

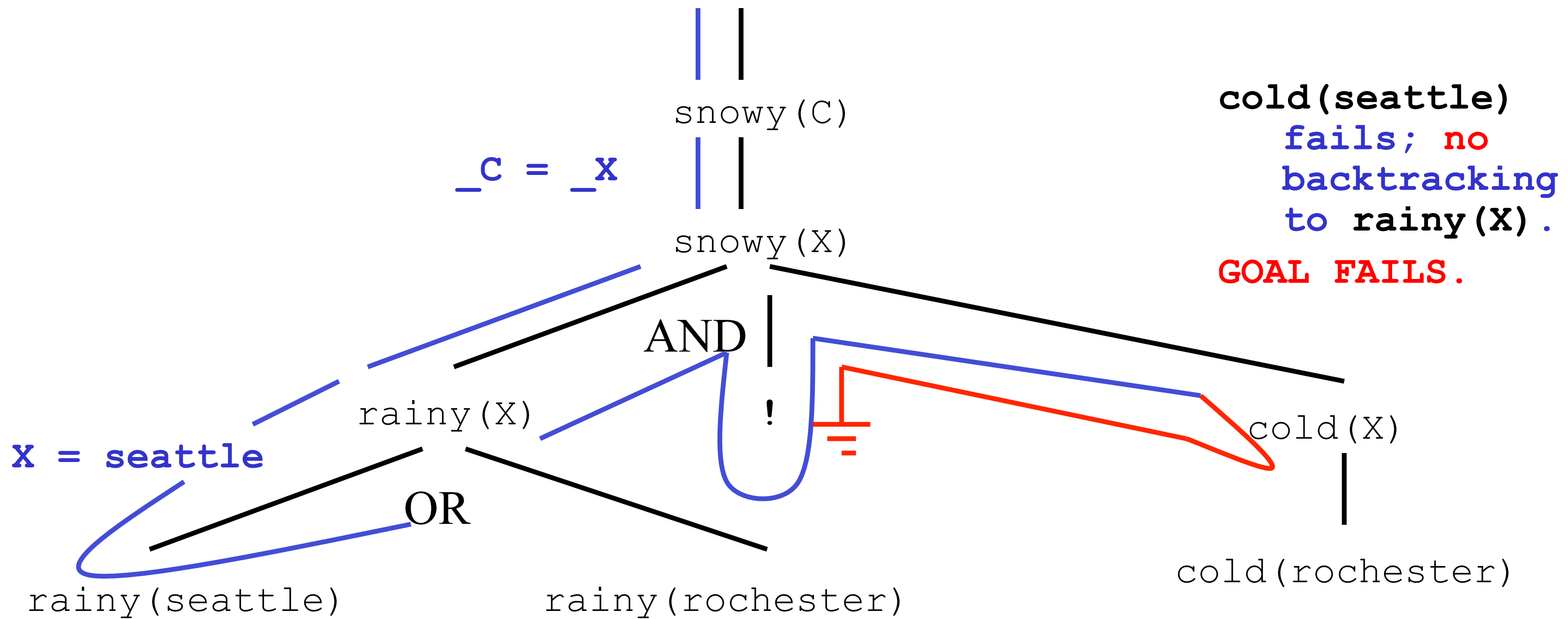
- As a goal it succeeds, but with a side effect:
  - Commits interpreter to choices made since unifying parent goal with left-hand side of current rule. Choices include variable unifications and rule to satisfy the parent goal.

Cut (!) Example

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).
```

Cut (!) Example

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).
```



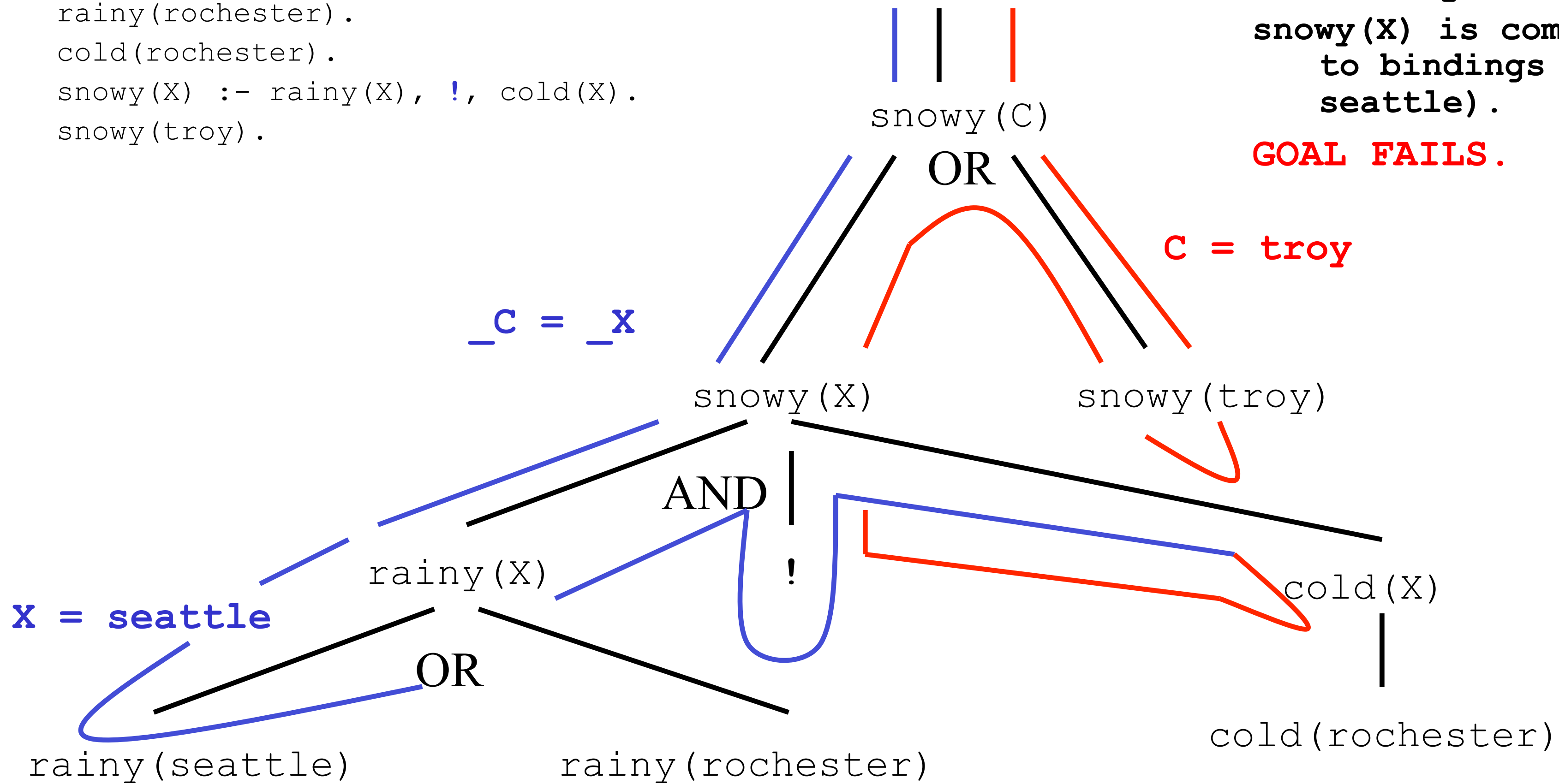
cold(seattle) **fails; no backtracking to rainy(X).**

GOAL FAILS.

Cut (!) Example 2

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), !, cold(X).
snowy(troy).
```

Cut (!) Example 2



C = troy FAILS

snowy(X) is committed to bindings (X = seattle).

GOAL FAILS.

Cut (!) Example 3

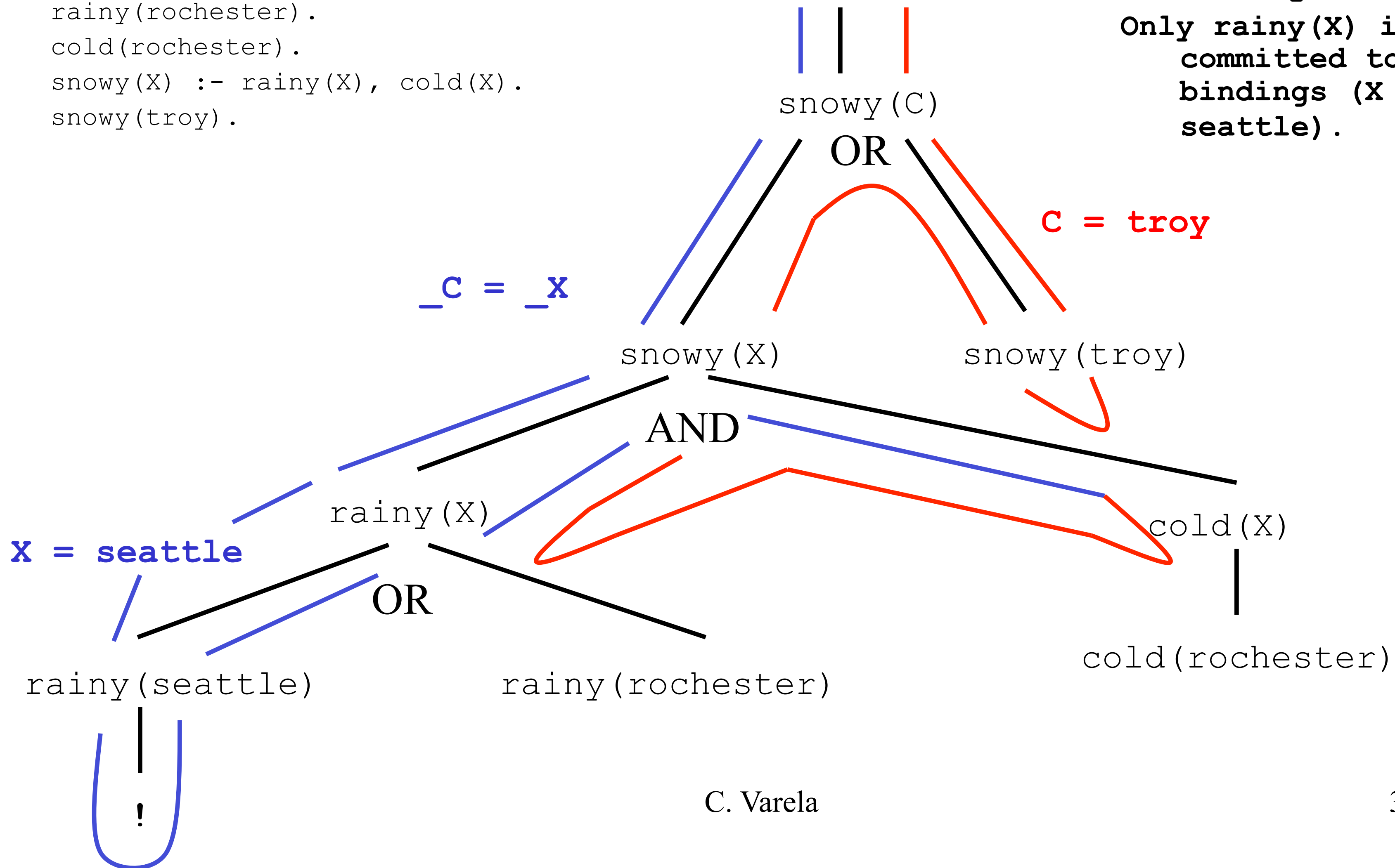
```
rainy(seattle) :- !.
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
snowy(troy).
```

Cut (!) Example 3

```
rainy(seattle) :- !.
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
snowy(troy).
```

C = troy SUCCEEDS

Only rainy(X) is committed to bindings (X = seattle).

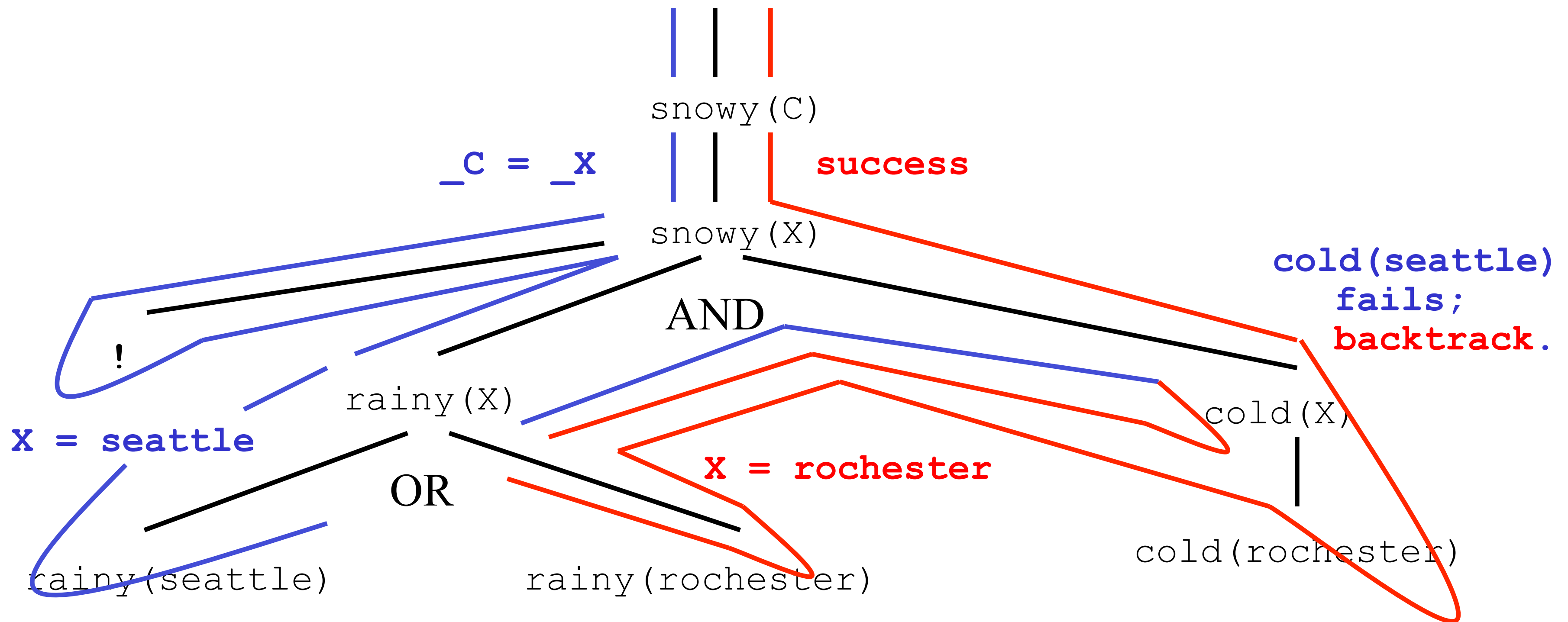


Cut (!) Example 4

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- !, rainy(X), cold(X).
```

Cut (!) Example 4

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- !, rainy(X), cold(X).
```

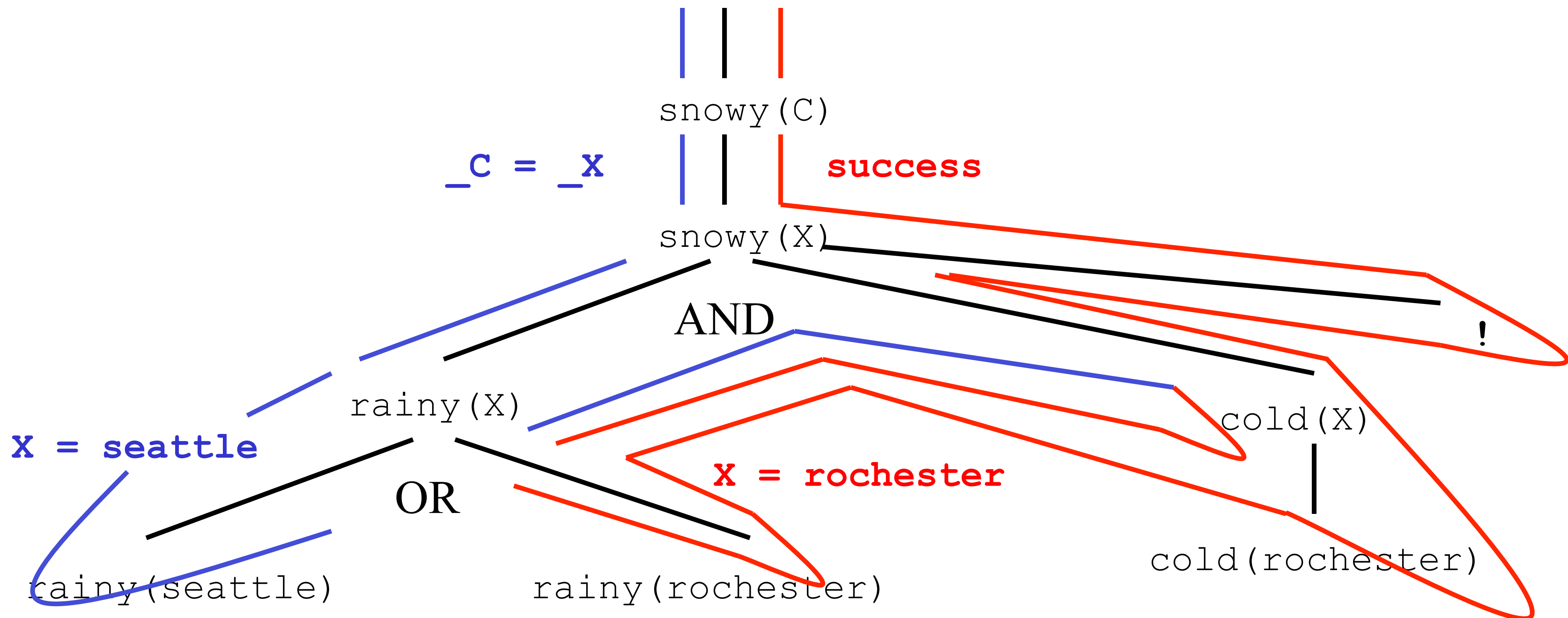


Cut (!) Example 5

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X), !.
```

Cut (!) Example 5

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X), !.
```



First-Class Terms

`call(P)`	Invoke predicate as a goal.
`assert(P)`	Adds predicate to database.
`retract(P)`	Removes predicate from database.
`functor(T,F,A)`	Succeeds if `T` is a *term* with *functor* `F` and *arity* `A`.
`findall(F,P,L)`	Returns a list L with elements F satisfying predicate P

`not P` is not ¬P

- In Prolog, the database of facts and rules includes a list of things assumed to be **true**.
- It does not include anything assumed to be **false**.
- Unless our database contains everything that is **true** (the *closed-world assumption*), the goal `not P` (or `\+ P` in some Prolog implementations) can succeed simply because our current knowledge is insufficient to prove `P`.

More `not` vs ¬

```
?- snowy(X).
X = rochester
?- not(snowy(X)).
no
```

Prolog does not reply: **`X = seattle.`**

The meaning of `not(snowy(X))` is:

$\neg \exists X \; [snowy(X)]$

rather than:

$\exists X \; [\neg snowy(X)]$

Fail, true, repeat

`fail`	Fails current goal.
`true`	Always succeeds.
`repeat`	Always succeeds, provides infinite choice points.

```
repeat.
repeat :- repeat.
```

`not` Semantics

```
not(P) :- call(P), !, fail.
not(P).
```

Definition of `not` in terms of failure (`fail`) means that variable bindings are lost whenever `not` succeeds, e.g.:

```
?- not(not(snowy(X))).
X=_G147
```

Conditionals and Loops

```
statement :- condition, !, then.
statement :- else.


natural(1).
natural(N) :-  natural(M), N is M+1.
my_loop(N) :-  N>0,
               natural(I),
               write(I), nl,
               I=N,
               !, fail.
```

Also called *generate-and-test*.

Prolog lists

- `[a,b,c]` is syntactic sugar for:

  `.(a,.(b,.(c, [])))`

  where `[]` is the empty list, and `.` is a built-in cons-like functor.

- `[a,b,c]` can also be expressed as:

  `[a | [b,c]]` , or
  `[a, b | [c]]` , or
  `[a,b,c | []]`

Prolog lists append example

```
append([],L,L).
append([H|T], A, [H|L]) :- append(T,A,L).
```

Oz lists (Review)

- `[a b c]` is syntactic sugar for:

```
'|'(a '|'(b '|'(c nil)))
```

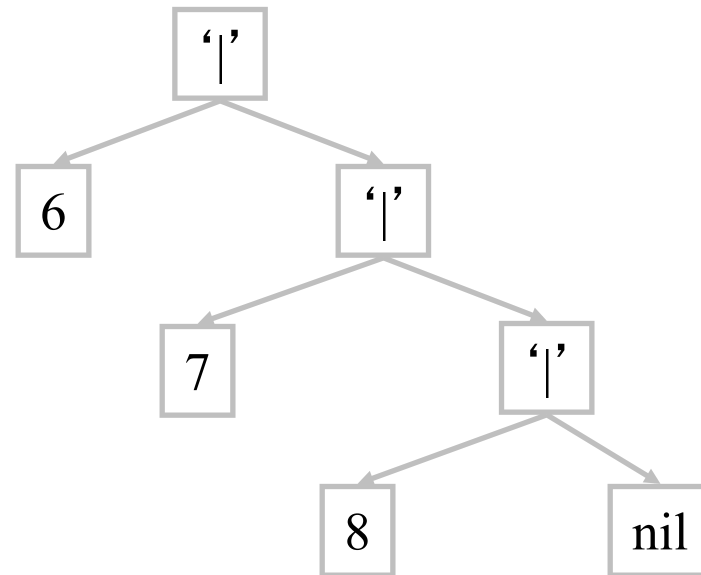
where `nil` is the empty list, and `'|'` is the tuple's functor.

- A list has two components:
  a head, and a tail

```
declare L = [6 7 8]
```

L.1 gives 6

L.2 give [7 8]



Oz lists append example

```
proc {Append Xs Ys Zs}
  choice Xs = nil Zs = Ys
  [] X Xr Zr in
    Xs=X|Xr
    Zs=X|Zr
    {Append Xr Ys Zr}
  end
end
```

```
% new search query
proc {P S}
    X Y in
    {Append X Y [1 2 3]} S=X#Y
end

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

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

% calculate all
{Browse {Search.base.all P}}
```

Arithmetic Goals

```
N>M
N<M
N=<M
N>=M
```

- `N` and `M` must be bound to numbers for these tests to *succeed* or *fail*.
- `X **is** 1+2` is used to *assign* numeric value of right-hand-side to variable in left-hand-side.

Loop Revisited

```
natural(1).
natural(N) :-  natural(M), N is M+1.
my_loop(N) :-  N>0,
               natural(I),
               write(I), nl,
               I=N,
               !.
my_loop(_).
```

Also called *generate-and-test*.

`=` is not equal to `==` or `=:=`

`X=Y` `X\=Y`

test whether `X` and `Y` **can be** or **cannot be** *unified*.

`X==Y` `X\==Y`

test whether `X` and `Y` are currently *co-bound*, i.e., have been bound to, or share the same value.

`X=:=Y` `X=\=Y`

test *arithmetic* equality and inequality.

Prolog Operators

```
:- op(P,T,O)
```

declares an operator symbol `O` with precedence `P` and type `T`.

- Example:

```
:- op(500,xfx,'has_color')
a has_color red.
b has_color blue.
```

then:

```
?- b has_color C.
C = blue.
?- What has_color red.
What = a.
```

Operator precedence/type

- Precendence **`P`** is an integer: the larger the number, the less the precedence (*ability to group*).
- Type **`T`** is one of:

`T`	**Position**	**Associativity**	**Examples**
`xfx`	Infix	Non-associative	`is`
`xfy`	Infix	Right-associative	`, ;`
`yfx`	Infix	Left-associative	`+ - * /`
`fx`	Prefix	Non-associative	`?-`
`fy`	Prefix	Right-associative	
`xf`	Postfix	Non-associative	
`yf`	Postfix	Left-associative	

Testing types

`atom``(X)`

tests whether `X` is an *atom*, e.g., `'foo'`, `bar`.

`integer``(X)`

tests whether `X` is an *integer*; it does not test for complex terms, e.g., `integer(4/2)` fails.

`float``(X)`

tests whether `X` is a *float*; it matches exact type.

`string``(X)`

tests whether `X` is a *string*, enclosed in `` ` ` `` ... `` ` ` ``.

Prolog Input

`seeing``(X)`

succeeds if `X` is (or can be) bound to *current read port*.

`X = user` is keyboard (standard input.)

`see``(X)`

opens port for input file bound to `X`, and makes it *current*.

`seen`

closes current port for input file, and makes `user` *current*.

`read``(X)`

reads Prolog type expression from *current* port, storing value in `X`.

`end-of-file`

is returned by **read** at <*end-of-file*>.

Prolog Output

`telling``(X)`

succeeds if `X` is (or can be) bound to *current output port*.

`X = user` is screen (standard output.)

`tell``(X)`

opens port for output file bound to `X`, and makes it *current*.

`told`

closes current output port, and reverses to screen output (makes `user` *current*.)

`write``(X)`

writes Prolog expression bound to `X` into *current* output port.

`nl`

new line (line feed).

`tab``(N)`

writes `N` spaces to current output port.

I/O Example

```
browse(File) :-
  seeing(Old),                /* save for later */
  see(File),                  /* open this file */
  repeat,
  read(Data),                 /* read from File */
  process(Data),
  seen,                       /* close File */
  see(Old),                   /* prev read source */
  !.                          /* stop now */

process(end_of_file) :- !.
process(Data) :-  write(Data), nl, fail.
```

Databases: assert and retract

- Prolog enables direct modification of its knowledge base using assert and retract.
- Let us consider a tic-tac-toe game:

1	2	3
4	5	6
7	8	9

- We can represent a board with facts x(n) and o(n), for n in {1..9} corresponding to each player's moves.
- As a player (or the computer) moves, a fact is dynamically added to Prolog's knowledge base.

Databases: assert and retract

```
% main goal:
play :- clear, repeat, getmove, respond.

getmove :-  repeat,
            write('Please enter a move: '),
            read(X), empty(X),
            assert(o(X)).

respond :- makemove, printboard, done.

makemove :- move(X), !, assert(x(X)).
makemove :- all_full.

clear :- retractall(x(_)), retractall(o(_)).
```

Human move

Computer move

Tic-tac-toe: Strategy

The strategy is to first try to win, then try to block a win, then try to create a split (forced win in the next move), then try to prevent opponent from building three in a row, and creating a split, finally choose center, corners, and other empty squares. The order of the rules is key to implementing the strategy.

```
move(A) :- good(A), empty(A), !.

good(A) :- win(A).
good(A) :- block_win(A).
good(A) :- split(A).
good(A) :- strong_build(A).
good(A) :- weak_build(A).

good(5).
good(1).  good(3).  good(7).  good(9).
good(2).  good(4).  good(6).  good(8).
```

Tic-tac-toe: Strategy(2)

o		
	x	o
		x

- Moving x(8) produces a split: x(2) or x(7) wins in next move.

```
win(A)        :- x(B), x(C), line(A,B,C).
block_win(A):- o(B), o(C), line(A,B,C).
split(A)      :- x(B), x(C), different(B,C),
   line(A,B,D), line(A,C,E), empty(D), empty(E).
strong_build(A) :- x(B), line(A,B,C), empty(C),
                   not(risky(C)).
weak_build(A)   :- x(B), line(A,B,C), empty(C),
                   not(double_risky(C)).

risky(C)      :- o(D), line(C,D,E), empty(E).
double_risky(C) :- o(D), o(E), different(D,E),
   line(C,D,F), line(C,E,G), empty(F), empty(G).
```

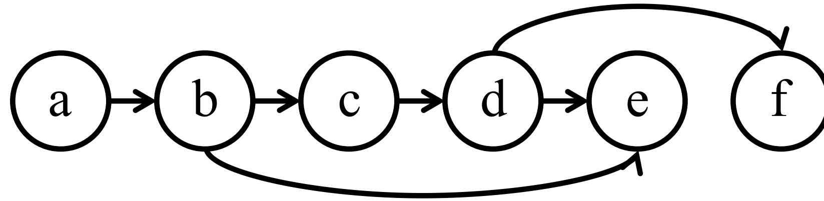
Databases in Oz: RelationClass

- Oz supports dynamic database modifications using a RelationClass. The initial relation is defined as follows:

  ```
  Rel = {New RelationClass init}
  ```

- Once we have a relation instance, the following operations are possible:
  - {Rel assert(T)} adds tuple T to Rel.
  - {Rel assertall(Ts)} adds the list of tuples Ts to Rel.
  - {Rel query(X)} binds X to one of the tuples in Rel. X can be any partial value. If more than one tuple is compatible with X, then search can enumerate all of them.

Databases in Oz: An example




```
GraphRel = {New RelationClass init}
{GraphRel assertall([edge(a b) edge(b c) edge(c d)
                     edge(d e) edge(b e) edge(d f)])}
proc {EdgeP A B} {GraphRel query(edge(A B))} end
{Browse {Search.base.all proc {$ X} {EdgeP b X} end}}
% displays all edges from node b: [c e]
```

Databases in Oz: An example(2)

```
proc {Path X Y}
   choice
      X = Y
   [] Z in
      {EdgeP Z Y}
      {Path X Z}
   end
end
{Browse {Search.base.all proc {$ X} {Path b X} end}}
% displays all nodes with a path from node b: [b c e e f d]
```

Natural Language Parsing

(Example from "Learn Prolog Now!" Online Tutorial)

```
word(article,a).
word(article,every).
word(noun,criminal).
word(noun,'big kahuna burger').
word(verb,eats).
word(verb,likes).

sentence(Word1,Word2,Word3,Word4,Word5) :-
      word(article,Word1),
      word(noun,Word2),
      word(verb,Word3),
      word(article,Word4),
      word(noun,Word5).
```

Parsing natural language

- *Definite Clause Grammars (DCG)* are useful for natural language parsing.
- Prolog can load DCG rules and convert them automatically to Prolog parsing rules.

DCG Syntax

`-->`

DCG *operator*, e.g.,

```
sentence-->subject,verb,object.
```

Each goal is assumed to refer to the *head* of a DCG rule.

`{prolog_code}`

Include Prolog code in generated parser, e.g.,

```
subject-->modifier,noun,{write('subject')}.
```

`[terminal_symbol]`

Terminal symbols of the grammar, e.g.,

```
noun-->[cat].
```

Natural Language Parsing

(example rewritten using DCG)

```
sentence --> article, noun, verb, article, noun.

article --> [a] | [every].

noun --> [criminal] | ['big kahuna burger'].

verb --> [eats] | [likes].
```

Natural Language Parsing (2)

```
sentence(V) --> subject, verb(V), subject.

subject --> article, noun.

article --> [a] | [every].

noun --> [criminal] | ['big kahuna burger'].

verb(eats) --> [eats].
verb(likes) --> [likes].
```

Accumulators

- *Accumulator programming* is a way to handle state in declarative programs. It is a programming technique that uses arguments to carry state, transform the state, and pass it to the next procedure.
- Assume that the state S consists of a number of components to be transformed individually:

  $S = (X,Y,Z,...)$
- For each predicate P, each state component is made into a pair, the first component is the *input* state and the second component is the output state after P has terminated
- S is represented as

  $(X_{in}, X_{out}, Y_{in}, Y_{out}, Z_{in}, Z_{out},...)$

A Trivial Example in Prolog

```
increment(N0,N) :-
    N is N0 + 1.

square(N0,N) :-
    N is N0 * N0.

inc_square(N0,N) :-
    increment(N0,N1),
    square(N1,N).
```

`increment` takes N0 as the input and produces N as the output by adding 1 to N0.

`square` takes N0 as the input and produces N as the output by multiplying N0 to itself.

`inc_square` takes N0 as the input and produces N as the output by using an intermediate variable N1 to carry N0+1 (the output of `increment`) and passing it as input to `square`. The pairs N0-N1 and N1-N are called *accumulators*.

A Trivial Example in Oz

```
proc {Increment N0 N}
    N = N0 + 1
end

proc {Square N0 N}
    N = N0 * N0
end

proc {IncSquare N0 N}
    N1 in
    {Increment N0 N1}
    {Square N1 N}
end
```

`Increment` takes N0 as the input and produces N as the output by adding 1 to N0.

`Square` takes N0 as the input and produces N as the output by multiplying N0 to itself.

`IncSquare` takes N0 as the input and produces N as the output by using an intermediate variable N1 to carry N0+1 (the output of **`Increment`**) and passing it as input to **`Square`**. The pairs N0-N1 and N1-N are called *accumulators*.

Accumulators

- Assume that the state *S* consists of a number of components to be transformed individually:

  $S = (X,Y,Z)$

- Assume P1 to Pn are procedures in Oz

accumulator

```
proc {P X0 X Y0 Y Z0 Z}
         :
   {P1 X0 X1 Y0 Y1 Z0 Z1}
   {P2 X1 X2 Y1 Y2 Z1 Z2}
         :
   {Pn Xn-1 X Yn-1 Y Zn-1 Z}
end
```

The same concept applies to predicates in Prolog

- The procedural syntax is easier to use if there is more than one accumulator

MergeSort Example

- Consider a variant of MergeSort with accumulator
- proc {MergeSort1 N S0 S Xs}
  - N is an integer,
  - S0 is an input list to be sorted
  - S is the remainder of S0 after the first N elements are sorted
  - Xs is the sorted first N elements of S0
- The pair (S0, S) is an accumulator
- The definition is in a procedural syntax in Oz because it has two outputs S and Xs

Example (2)

```
fun {MergeSort Xs}
  Ys in
  {MergeSort1 {Length Xs} Xs _ Ys}
  Ys
end
```

```
proc {MergeSort1 N S0 S Xs}
  if N==0 then S = S0 Xs = nil
  elseif N ==1 then X in X|S = S0
    Xs=[X]
  else %% N > 1
    local S1 Xs1 Xs2 NL NR in
      NL = N div 2
      NR = N - NL
      {MergeSort1 NL S0 S1 Xs1}
      {MergeSort1 NR S1 S Xs2}
      Xs = {Merge Xs1 Xs2}
    end
  end
end
```

MergeSort Example in Prolog

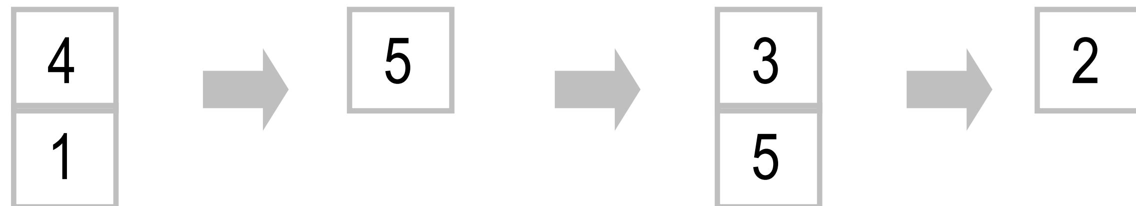
```
mergesort(Xs,Ys) :-
  length(Xs,N),
  mergesort1(N,Xs,_,Ys).
```

```
mergesort1(0,S,S,[]) :- !.
mergesort1(1,[X|S],S,[X]) :- !.
mergesort1(N,S0,S,Xs) :-
      NL is N // 2,
       NR is N - NL,
       mergesort1(NL,S0,S1,Xs1),
       mergesort1(NR,S1,S,Xs2),
       merge(Xs1,Xs2,Xs).
```

Multiple accumulators

- Consider a stack machine for evaluating arithmetic expressions
- Example: (1+4)-3
- The machine executes the following instructions

push(1)
push(4)
plus
push(3)
minus



Multiple accumulators (2)

- Example: (1+4)-3
- The arithmetic expressions are represented as trees:

  `minus(plus(1 4) 3)`
- Write a procedure that takes arithmetic expressions represented as trees and output a list of stack machine instructions and counts the number of instructions

```
proc {ExprCode Expr Cin Cout Nin Nout}
```

- Cin: initial list of instructions
- Cout: final list of instructions
- Nin: initial count
- Nout: final count

Multiple accumulators (3)

```
proc {ExprCode Expr C0 C N0 N}
  case Expr
  of plus(Expr1 Expr2) then C1 N1 in
    C1 = plus|C0
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] minus(Expr1 Expr2) then C1 N1 in
    C1 = minus|C0
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] I andthen {IsInt I} then
    C = push(I)|C0
    N = N0 + 1
  end
end
```

Multiple accumulators (4)

```
proc {ExprCode Expr C0 C N0 N}
  case Expr
  of plus(Expr1 Expr2) then C1 N1 in
    C1 = plus|C0
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] minus(Expr1 Expr2) then C1 N1 in
    C1 = minus|C0
    N1 = N0 + 1
    {SeqCode [Expr2 Expr1] C1 C N1 N}
  [] I andthen {IsInt I} then
    C = push(I)|C0
    N = N0 + 1
  end
end
```

```
proc {SeqCode Es C0 C N0 N}
  case Es
  of nil then C = C0 N = N0
  [] E|Er then N1 C1 in
     {ExprCode E C0 C1 N0 N1}
     {SeqCode Er C1 C N1 N}
  end
end
```

Shorter version (4)

```
proc {ExprCode Expr C0 C N0 N}
  case Expr
  of plus(Expr1 Expr2) then
    {SeqCode [Expr2 Expr1] plus|C0 C N0 + 1 N}
  [] minus(Expr1 Expr2) then
    {SeqCode [Expr2 Expr1] minus|C0 C N0 + 1 N}
  [] I andthen {IsInt I} then
    C = push(I)|C0
    N = N0 + 1
  end
end
```

```
proc {SeqCode Es C0 C N0
    N}
  case Es
  of nil then C = C0 N = N0
  [] E|Er then N1 C1 in
    {ExprCode E C0 C1 N0
    N1}
    {SeqCode Er C1 C N1 N}
  end
end
```

Functional style (4)

```
fun {ExprCode Expr t(C0 N0) }
  case Expr
  of plus(Expr1 Expr2) then
    {SeqCode [Expr2 Expr1] t(plus|C0 N0 + 1)}
  [] minus(Expr1 Expr2) then
    {SeqCode [Expr2 Expr1] t(minus|C0 N0 + 1)}
  [] I andthen {IsInt I} then
    t(push(I)|C0 N0 + 1)
  end
end
```

```
fun {SeqCode Es T}
  case Es
  of nil then T
  [] E|Er then
    T1 = {ExprCode E T} in
    {SeqCode Er T1}
  end
end
```

Difference lists in Oz

- A *difference list* is a pair of lists, each might have an unbound tail, with the invariant that one can get the second list by removing zero or more elements from the first list
- X # X % Represent the empty list
- nil # nil % idem
- [a] # [a] % idem
- (a|b|c|X) # X % Represents [a b c]
- [a b c d] # [d] % idem
- [a b c d|Y] # [d|Y] % idem
- [a b c d|Y] # Y % Represents [a b c d]

Difference lists in Prolog

- A *difference list* is a pair of lists, each might have an unbound tail, with the invariant that one can get the second list by removing zero or more elements from the first list
- X , X % Represent the empty list
- [] , [] % idem
- [a] , [a] % idem
- [a,b,c|X] , X % Represents [a,b,c]
- [a,b,c,d] , [d] % idem
- [a,b,c,d|Y] , [d|Y] % idem
- [a,b,c,d|Y] , Y % Represents [a,b,c,d]

Difference lists in Oz (2)

- When the second list is unbound, an append operation with another difference list takes constant time
- ```
  fun {AppendD D1 D2}
        S1 # E1 = D1
        S2 # E2 = D2
  in    E1 = S2
        S1 # E2
  end
  ```
- `local X Y in {Browse {AppendD (1|2|3|X)#X (4|5|Y)#Y}} end`
- Displays (1|2|3|4|5|Y)#Y
  ```

Difference lists in Prolog (2)

- When the second list is unbound, an append operation with another difference list takes constant time

```
append_dl(S1,E1, S2,E2, S1,E2) :- E1 = S2.
```

- ?- append_dl([1,2,3|X],X, [4,5|Y],Y, S,E).

Displays

X = [4, 5|_G193]

Y = _G193

S = [1, 2, 3, 4, 5|_G193]

E = _G193 ;

A FIFO queue with difference lists (1)

- A *FIFO queue* is a sequence of elements with an insert and a delete operation.
  - Insert adds an element to the end and delete removes it from the beginning
- Queues can be implemented with lists. If L represents the queue content, then deleting X can remove the head of the list matching X|T but inserting X requires traversing the list {Append L [X]} (insert element at the end).
  - Insert is inefficient: it takes time proportional to the number of queue elements
- With difference lists we can implement a queue with constant-time insert and delete operations
  - The queue content is represented as q(N S E), where N is the number of elements and S#E is a difference list representing the elements

A FIFO queue with difference lists (2)

```
fun {NewQueue} X in q(0 X X) end

fun {Insert Q X}
    case Q of q(N S E) then E1 in E=X|E1 q(N+1 S
    E1) end
end

fun {Delete Q X}
    case Q of q(N S E) then S1 in X|S1=S q(N-1 S1 E)
    end
end

fun {EmptyQueue Q} case Q of q(N S E) then N==0
    end end
```

- Inserting ‘b’:
  - In: q(1 a|T T)
  - Out: q(2 a|b|U U)
- Deleting X:
  - In: q(2 a|b|U U)
  - Out: q(1 b|U U) and X=a
- Difference list allows operations at both ends
- N is needed to keep track of the number of queue elements

Flatten

```
fun {Flatten Xs}
case Xs
  of nil then nil
  [] X|Xr andthen {IsLeaf X} then
    X|{Flatten Xr}
  [] X|Xr  andthen {Not {IsLeaf X}} then
     {Append {Flatten X} {Flatten Xr}}
  end
end
```

Flatten takes a list of elements and sub-lists and returns a list with only the elements, e.g.:

{Flatten [1 [2] [[3]]]} = [1 2 3]

Let us replace lists by difference lists and see what happens.

Flatten with difference lists (1)

- Flatten of nil is X#X
- Flatten of a leaf X|Xr is (X|Y1)#Y
  - flatten of Xr is Y1#Y
- Flatten of X|Xr is Y1#Y where
  - flatten of X is Y1#Y2
  - flatten of Xr is Y3#Y
  - equate Y2 and Y3

Here is what it looks like as text

Flatten with difference lists (2)

```
proc {FlattenD Xs Ds}
  case Xs
  of nil then Y in Ds = Y#Y
   [] X|Xr andthen {IsLeaf X} then Y1 Y in
      {FlattenD Xr Y1#Y2}
      Ds = (X|Y1)#Y
   [] X|Xr  andthen {IsList X} then Y0 Y1 Y2 in
      Ds = Y0#Y2
      {FlattenD X Y0#Y1}
      {FlattenD Xr Y1#Y2}
  end
end
fun {Flatten Xs}  Y in {FlattenD Xs Y#nil}  Y end
```

Here is the new program. It is much more efficient than the first version.

Reverse

- Here is our recursive reverse:

```
fun {Reverse Xs}
   case Xs
   of nil then nil
   [] X|Xr then {Append {Reverse Xr} [X]}
   end
end
```

- Rewrite this with difference lists:
  - Reverse of nil is X#X
  - Reverse of X|Xs is Y1#Y, where
    - reverse of Xs is Y1#Y2, and
    - equate Y2 and X|Y

Reverse with difference lists (1)

- The naive version takes time proportional to the square of the input length
- Using difference lists in the naive version makes it linear time
- We use two arguments Y1 and Y instead of Y1#Y
- With a minor change we can make it iterative as well

```
fun {ReverseD Xs}
    proc {ReverseD Xs Y1 Y}
        case Xs
        of nil then Y1=Y
        [] X|Xr then Y2 in
            {ReverseD Xr Y1 Y2}
            Y2 = X|Y
        end
    end
    R in
    {ReverseD Xs R nil}
    R
end
```

Reverse with difference lists (2)

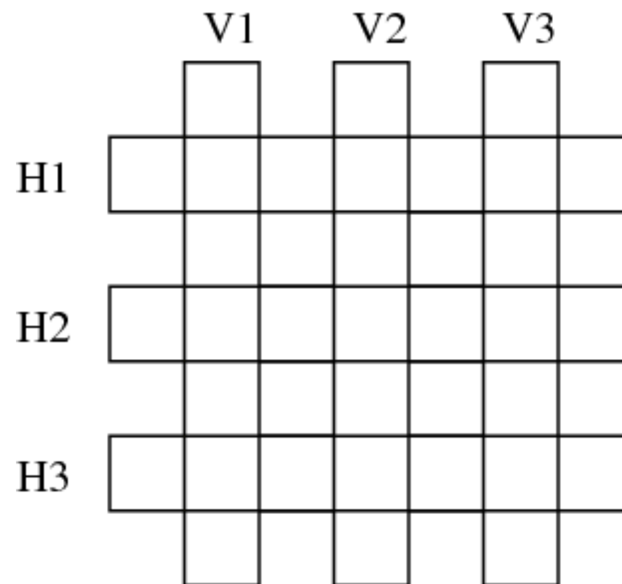
```
fun {ReverseD Xs}
    proc {ReverseD Xs Y1 Y}
        case Xs
        of nil then Y1=Y
        [] X|Xr then
            {ReverseD Xr Y1 X|Y}
        end
    end
    R in
    {ReverseD Xs R nil}
    R
end
```

Difference lists: Summary

- Difference lists are a way to represent lists in the declarative model such that one append operation can be done in constant time
  - A function that builds a big list by concatenating together lots of little lists can usually be written efficiently with difference lists
  - The function can be written naively, using difference lists and append, and will be efficient when the append is expanded out
- Difference lists are declarative, yet have some of the power of destructive assignment
  - Because of the single-assignment property of dataflow variables
- Difference lists originated from Prolog and are used to implement, e.g., definite clause grammar rules for natural language parsing.

Constraint Satisfaction Example*

- Given six Italian words:
  - astante , astoria , baratto , cobalto , pistola , statale .
- They are to be arranged, crossword puzzle fashion, in the following grid:



Constraint Satisfaction Example(2)*

- The following knowledge base represents a lexicon containing these words:

```
word(astante,  a,s,t,a,n,t,e).
word(astoria,  a,s,t,o,r,i,a).
word(baratto,  b,a,r,a,t,t,o).
word(cobalto,  c,o,b,a,l,t,o).
word(pistola,  p,i,s,t,o,l,a).
word(statale,  s,t,a,t,a,l,e).
```

- Write a predicate `crossword/6` that tells us how to fill in the puzzle. The first three arguments should be the vertical words from left to right, and the last three arguments the horizontal words from top to bottom.

Constraint Satisfaction Example(3)*

- Try solving it yourself before looking at this solution!

```
crossword(V1,V2,V3,H1,H2,H3) :-
      word(V1,_,H1V1,_,H2V1,_,H3V1,_),
      word(V2,_,H1V2,_,H2V2,_,H3V2,_),
      word(V3,_,H1V3,_,H2V3,_,H3V3,_),
      word(H1,_,H1V1,_,H1V2,_,H1V3,_),
      word(H2,_,H2V1,_,H2V2,_,H2V3,_),
      word(H3,_,H3V1,_,H3V2,_,H3V3,_).
```

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 digit pairs that add to 10

- Using generate and test to do combinatorial search:

```
fun {PairAdd10}
   D1 D2 in
   D1 = {Digit}           % generate
   D2 = {Digit}           % generate
   D1+D2 = 10             % test
   D1#D2
end
{Browse {Search.base.all PairAdd10}}
% displays [1#9 2#8 3#7 4#6 5#5 6#4 7#3 8#2 9#1]
```

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:

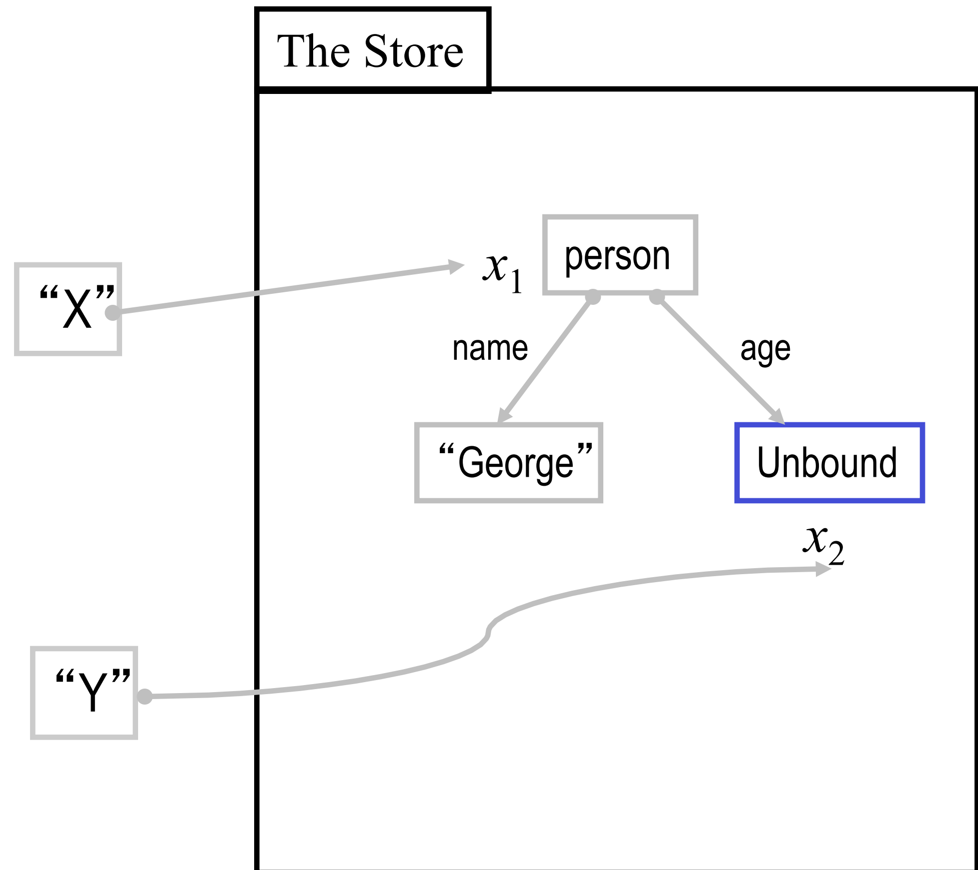
```
fun {Rectangle}
  W H in W::0#9   H::0#9
  W+H =: 10
  W*H >=: 24
  W <: H
  {FD.distribute naive rect(W H)}
  rect(W H)
end
{Browse {Search.base.all Rectangle}}
% displays [rect(4 6)]
```

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)

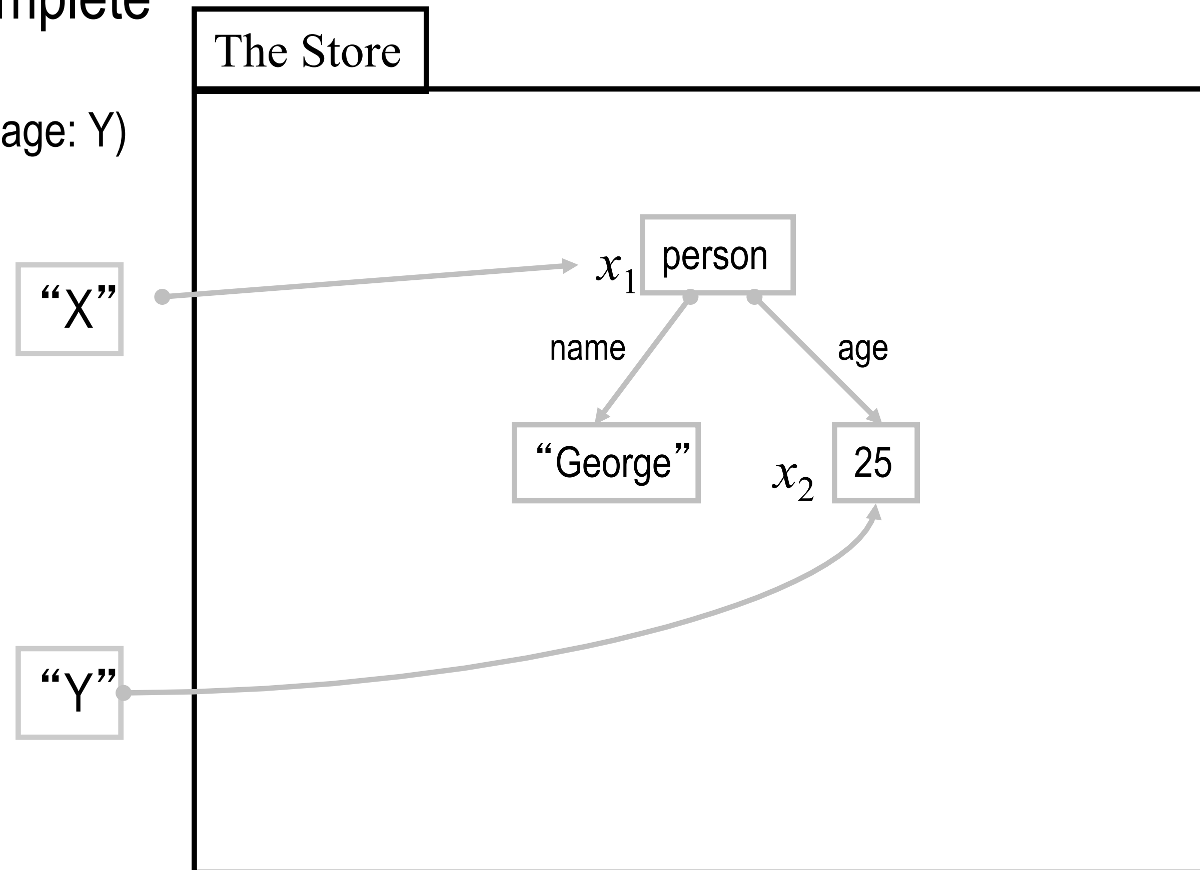
- Is a data structure that may contain unbound variables
- The store contains the partial value: person(name: "George" age: x_2)
- declare Y X
  X = person(name: "George" age: Y)
- The identifier 'Y' refers to x_2



Partial Values (2)

Partial Values may be complete

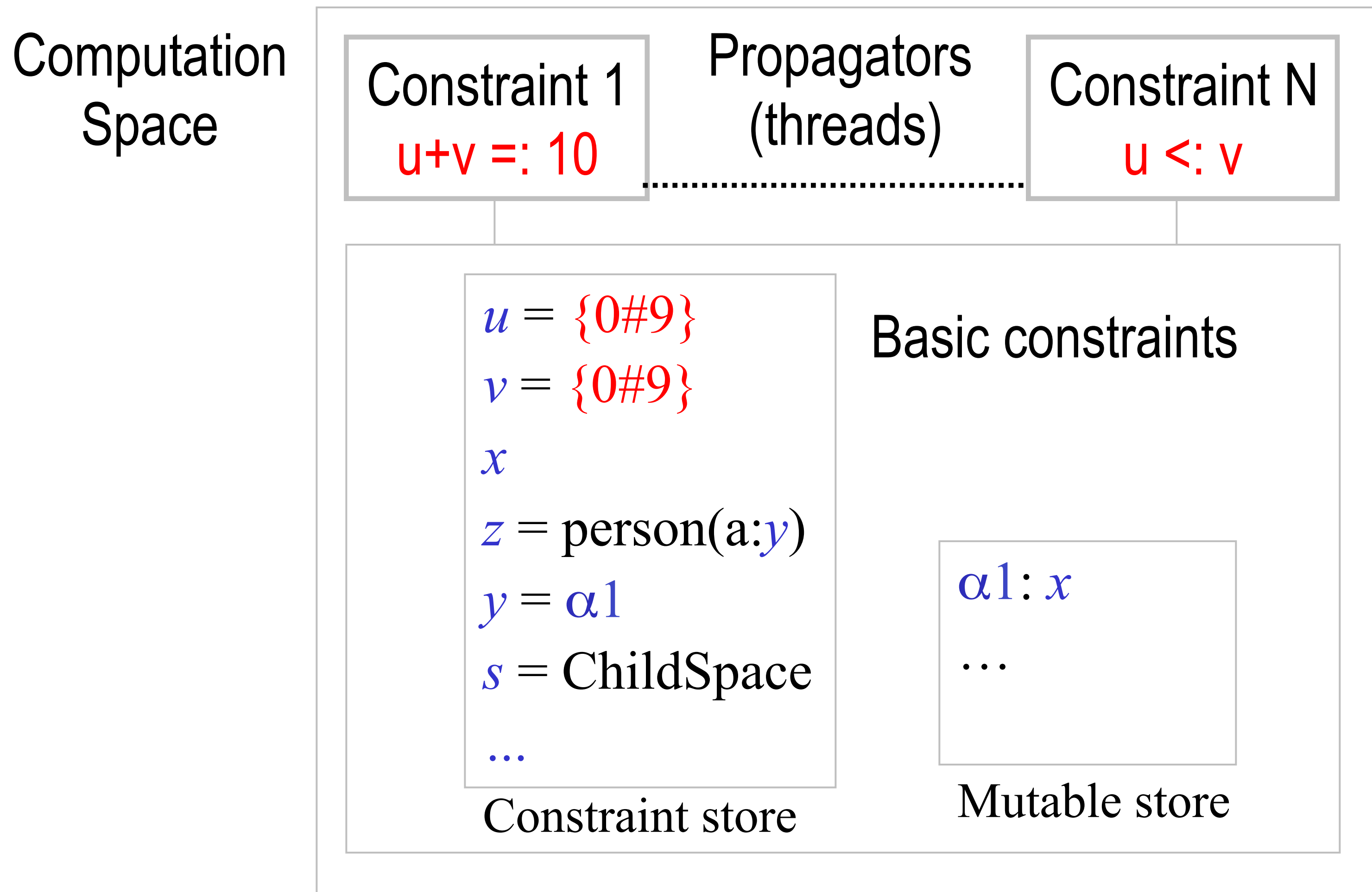
- declare Y X
  X = person(name: “George” age: Y)
- **Y = 25**



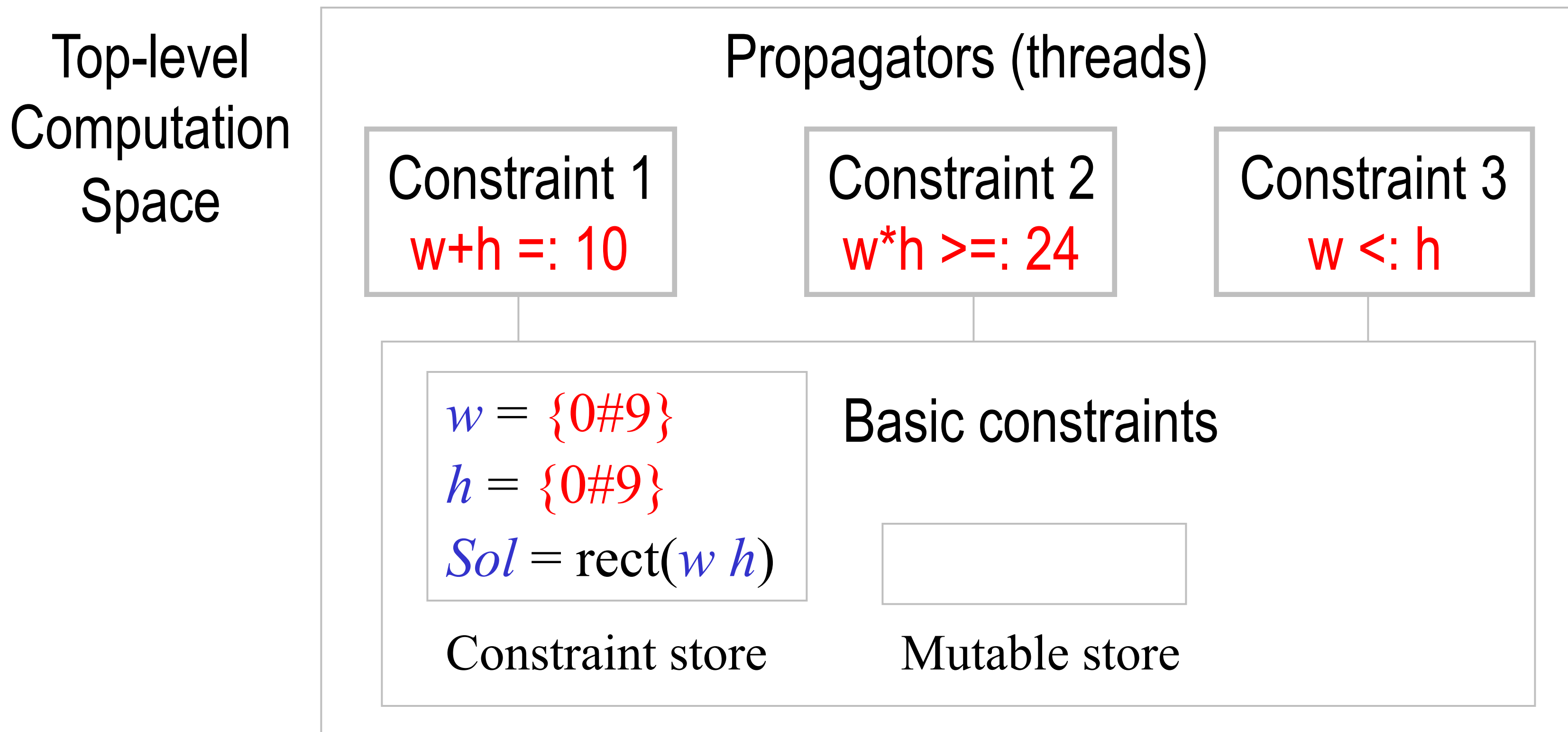
Variables and partial values

- 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

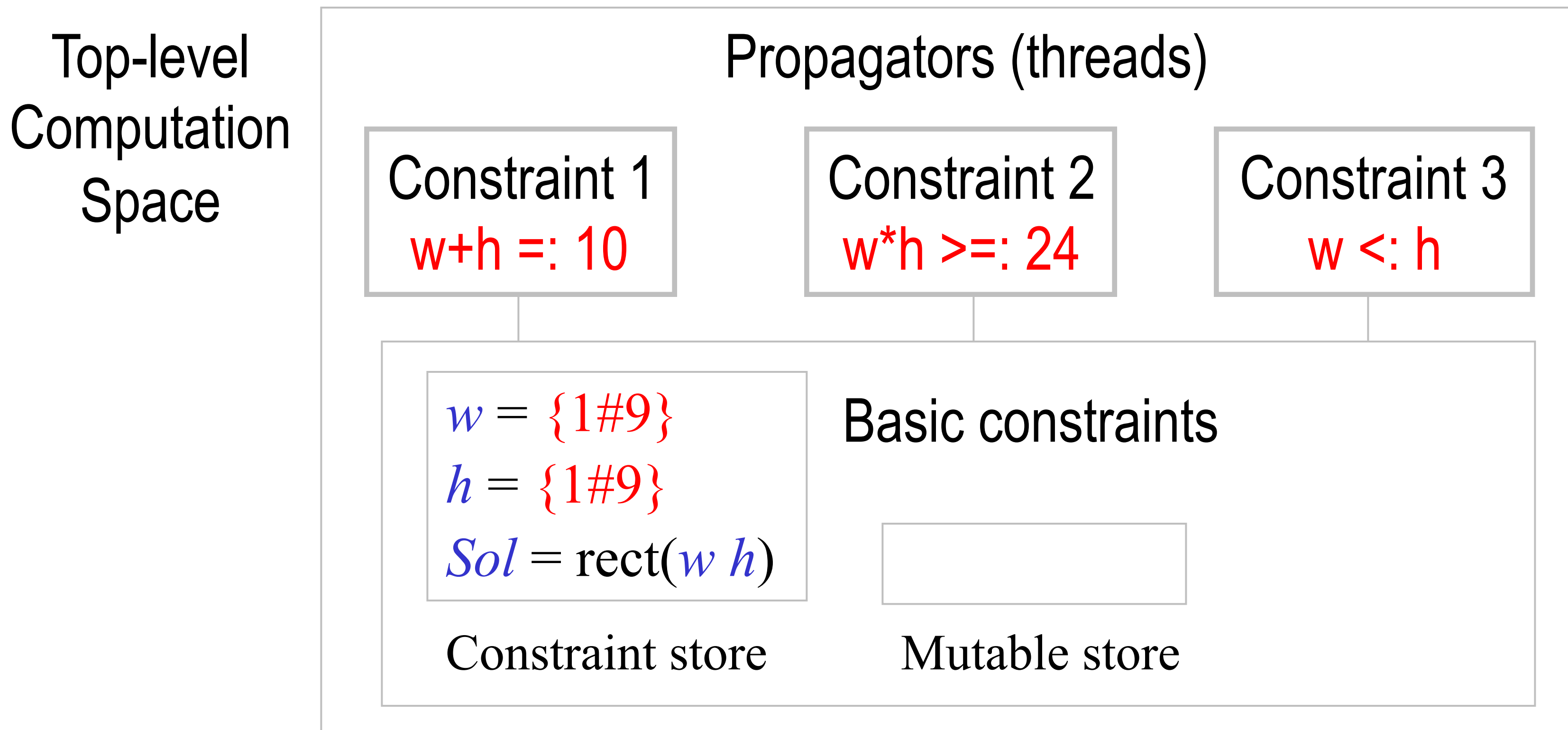


Constraint propagation: Rectangle example



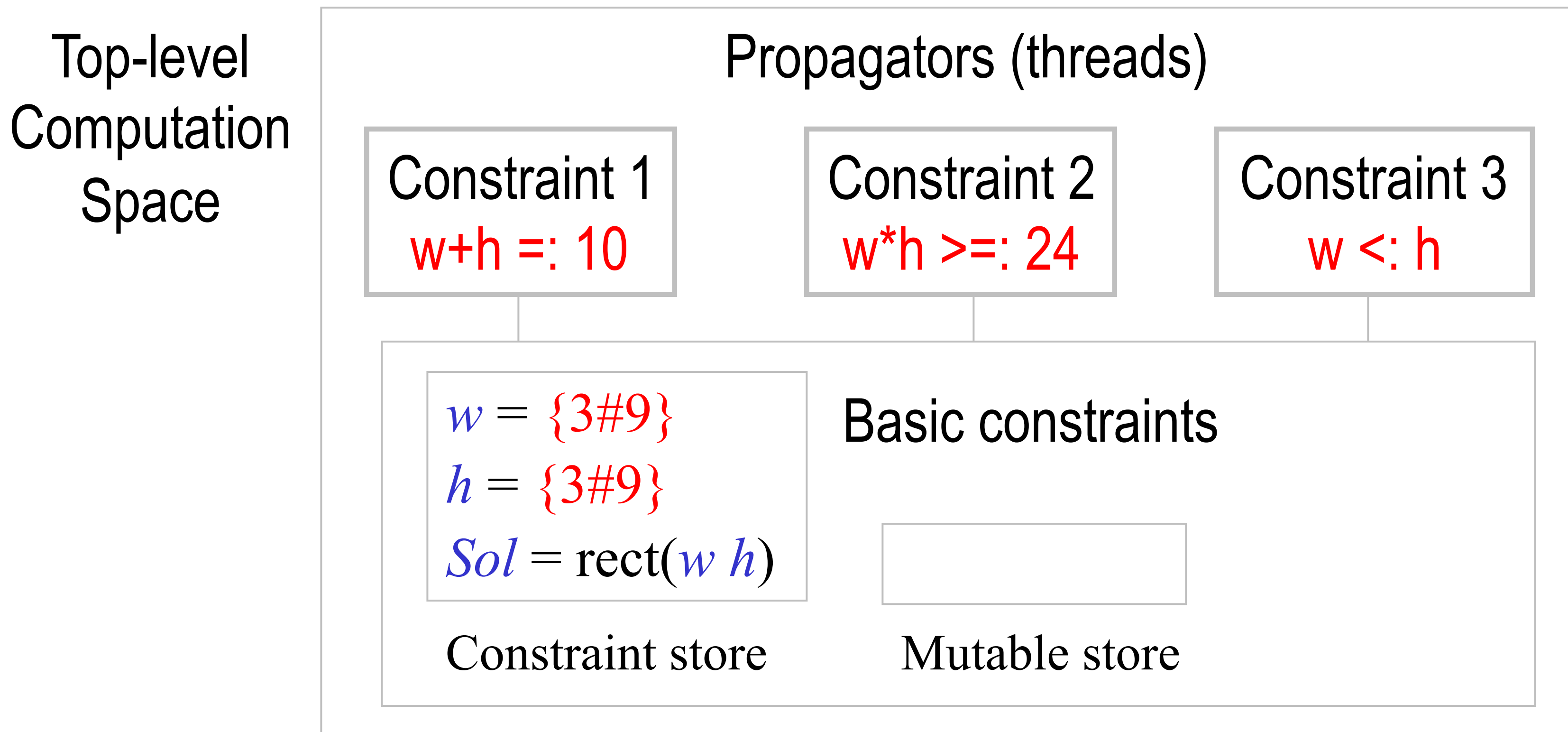
Let us consider propagator 1: w+h =: 10 → w cannot be 0; h cannot be 0.

Constraint propagation: Rectangle example



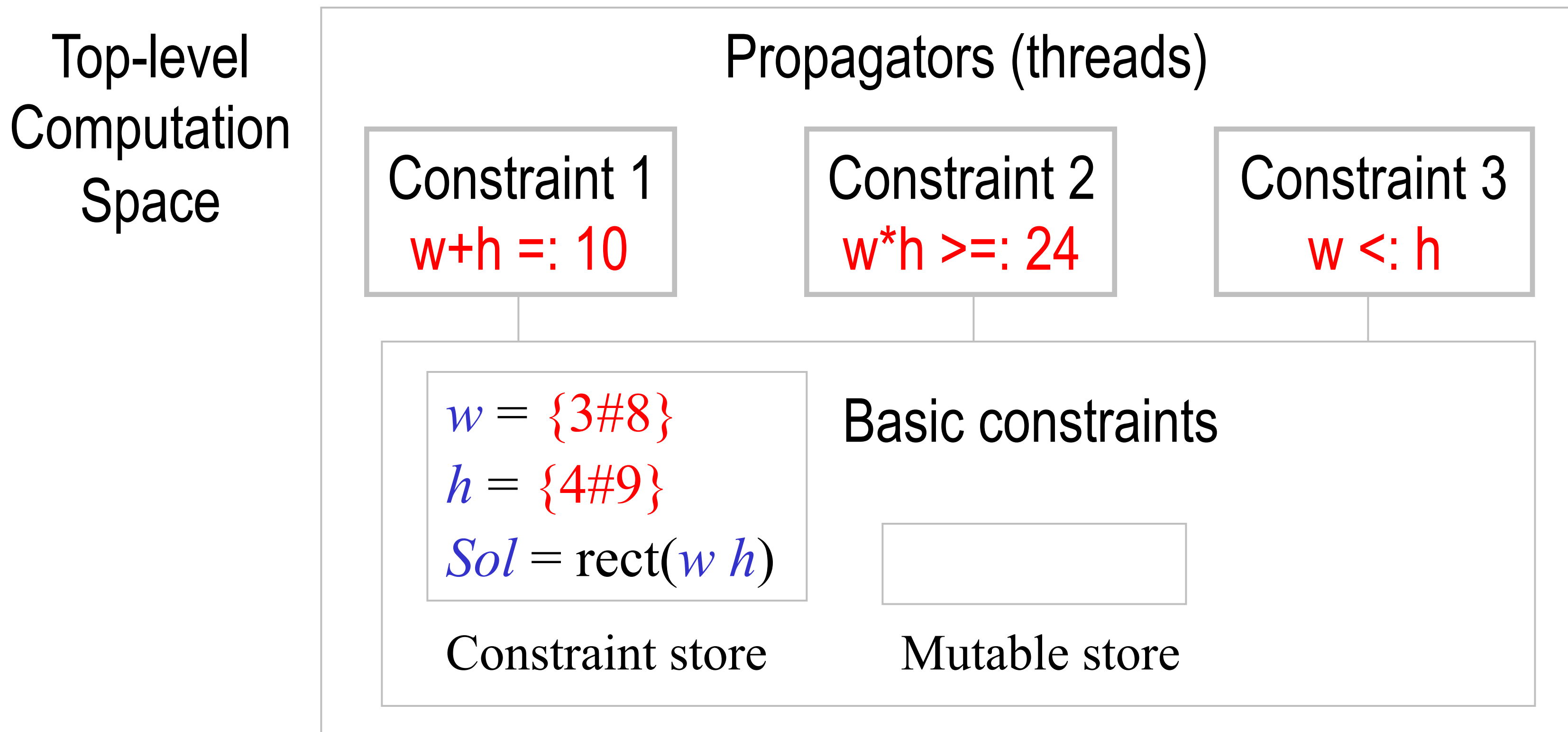
Let us consider propagator 2: w*h >=: 24 → w or h cannot be 1 or 2.

Constraint propagation: Rectangle example



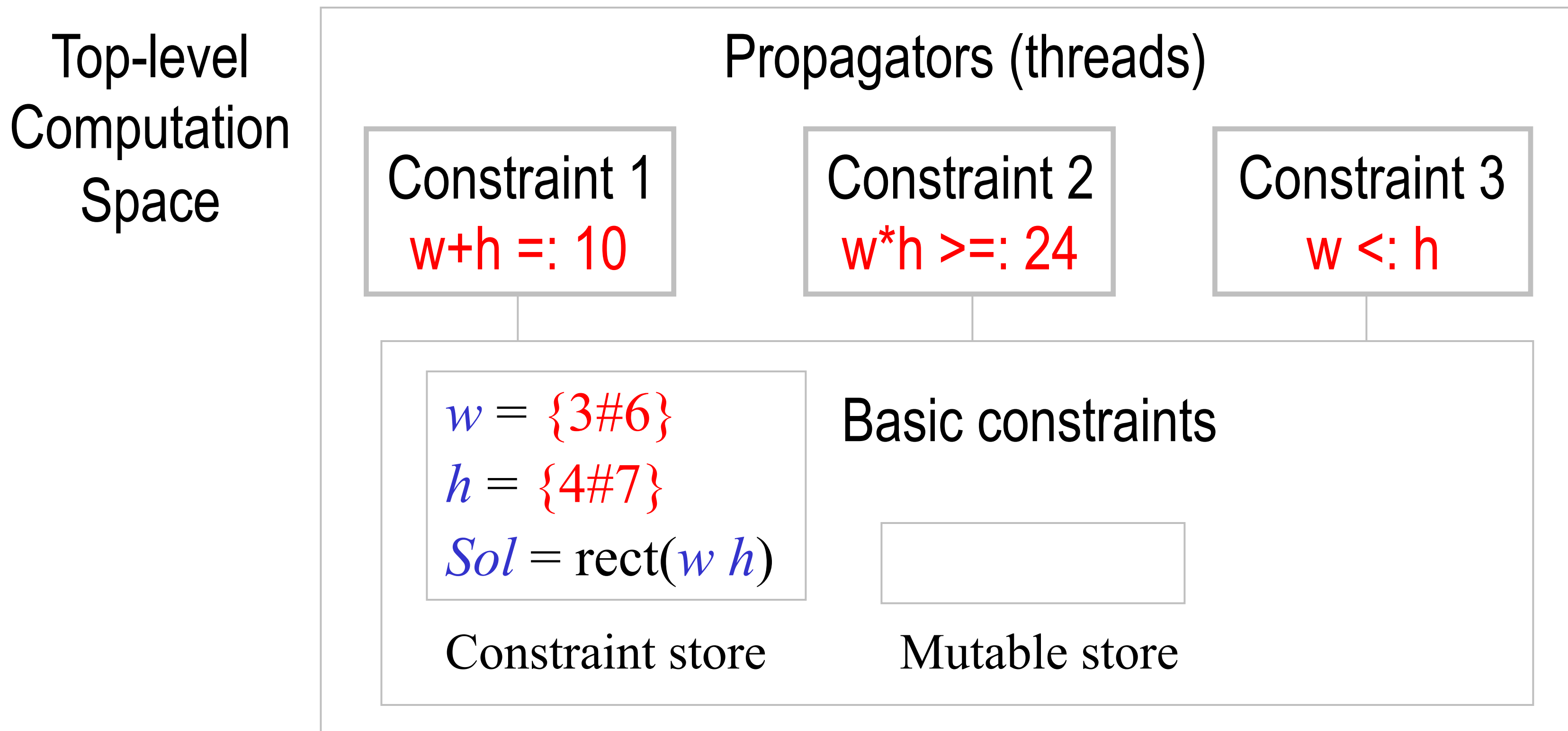
Let us consider propagator 3: w <: h → w cannot be 9, h cannot be 3.

Constraint propagation: Rectangle example



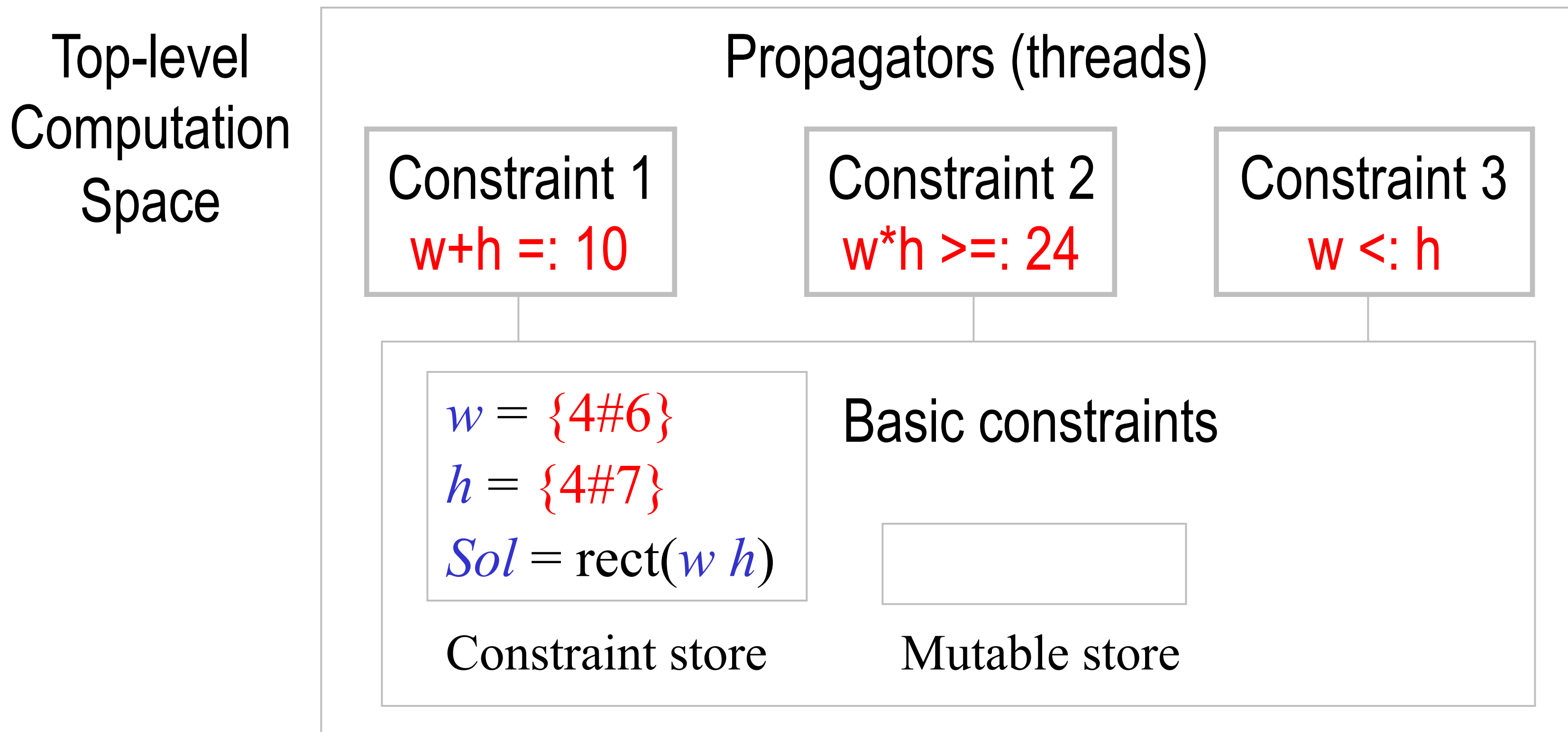
Let us consider propagator 1 **again**: w + h =: 10 →
w >= 3 implies h cannot be 8 or 9,
h >= 4 implies w cannot be 7 or 8.

Constraint propagation: Rectangle example



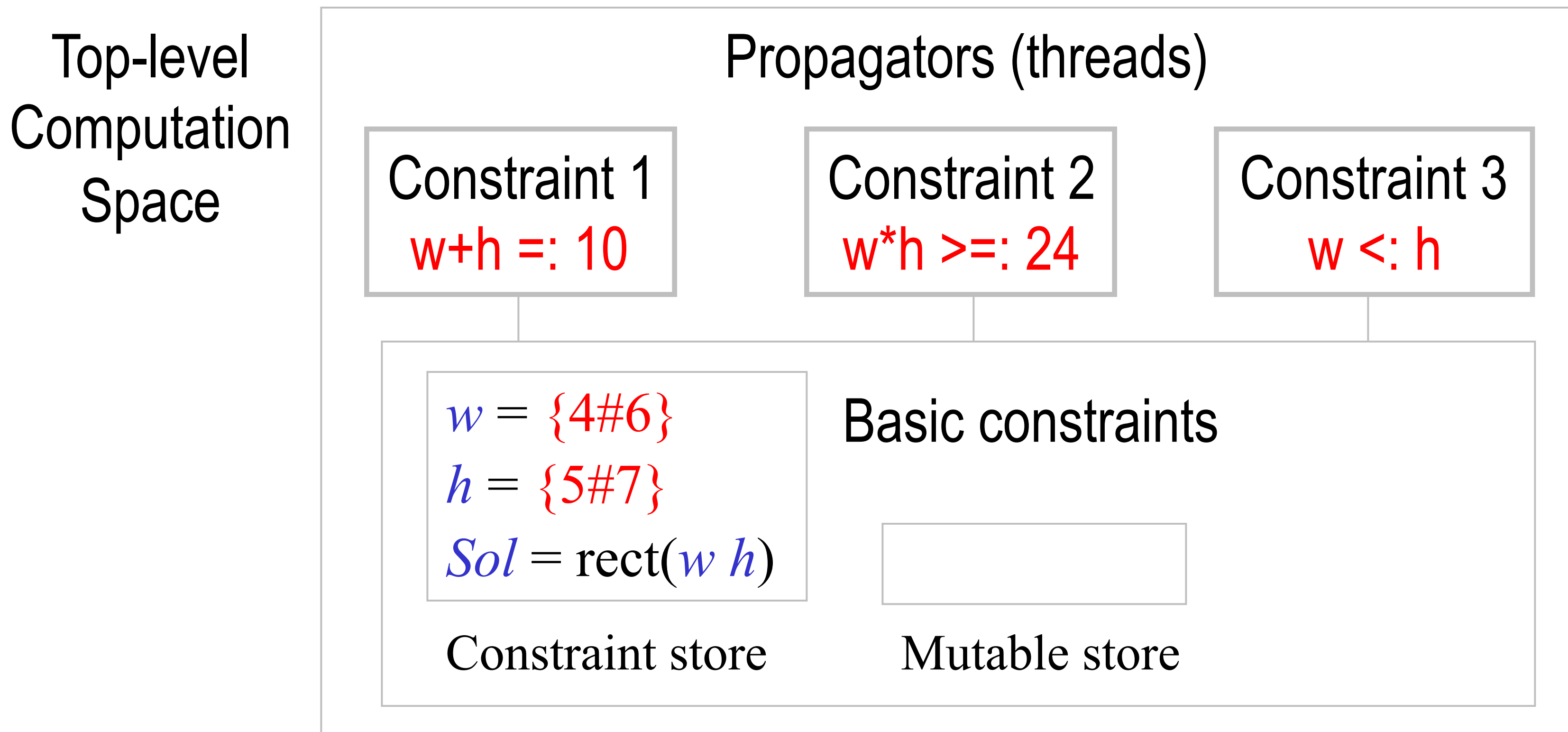
Let us consider propagator 2 **again**: w * h >=: 24 → h <= 7 implies w cannot be 3.

Constraint propagation: Rectangle example



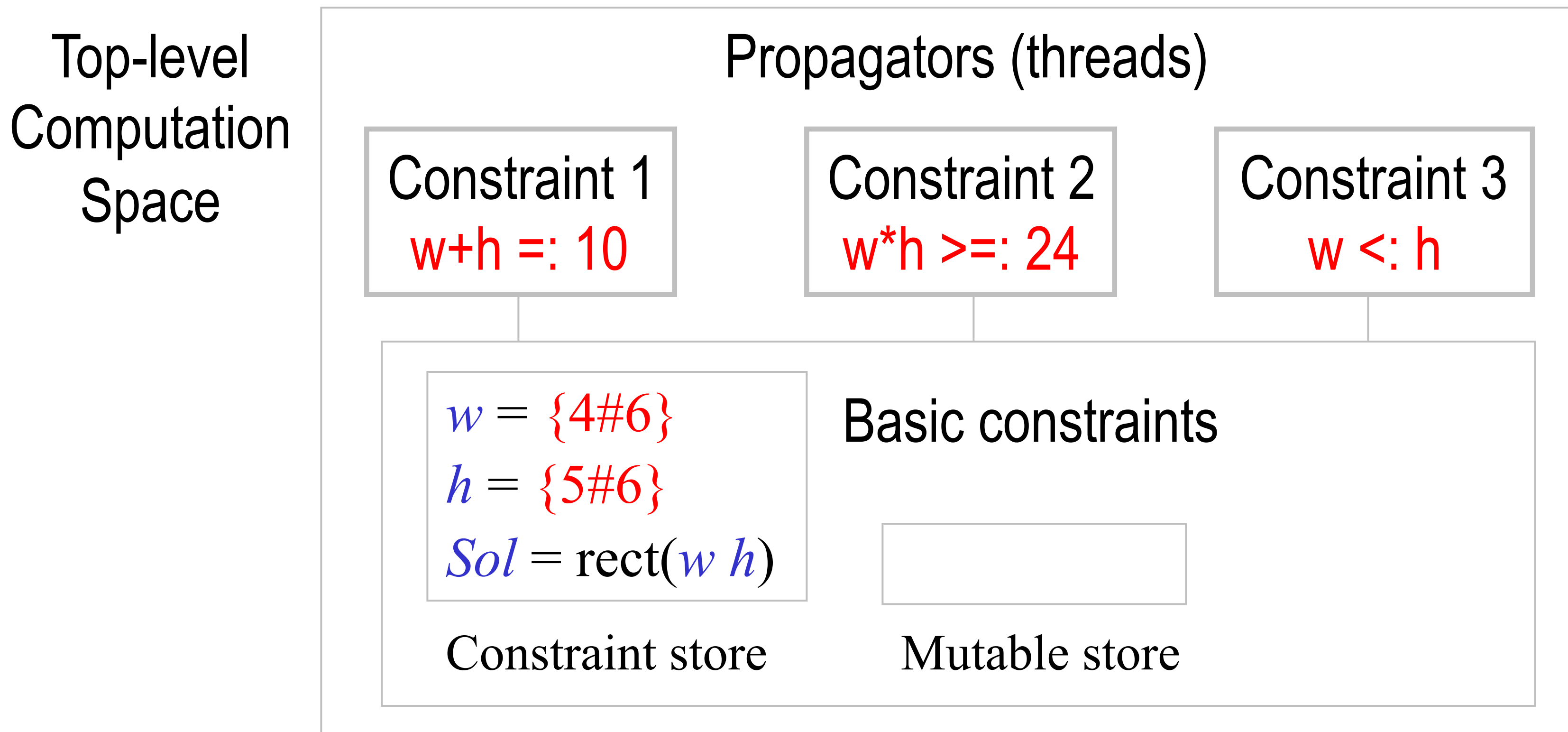
Let us consider propagator 3 **again**: w <: h → h cannot be 4.

Constraint propagation: Rectangle example



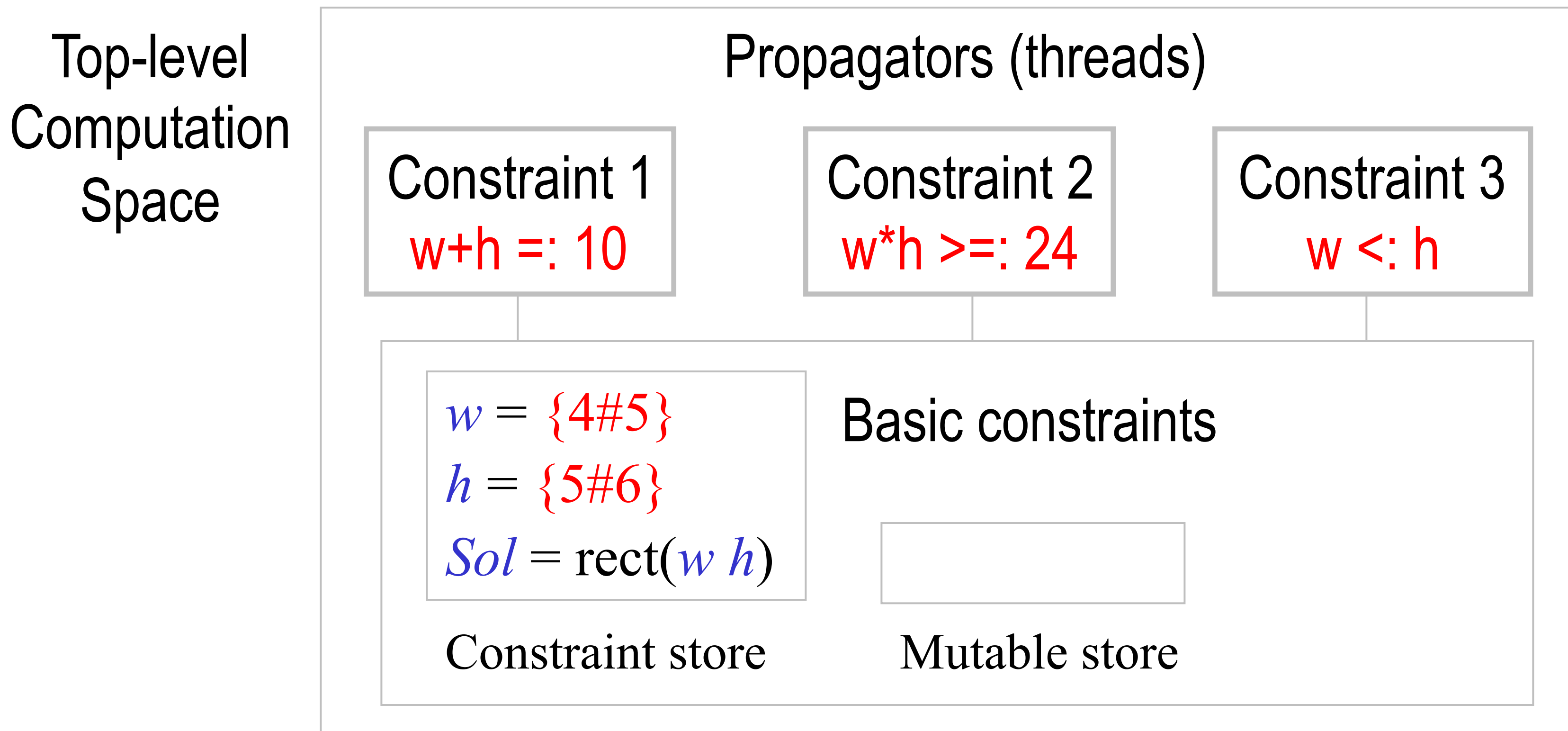
Let us consider propagator 1 **once more**: w + h =: 10 → h cannot be 7.

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.

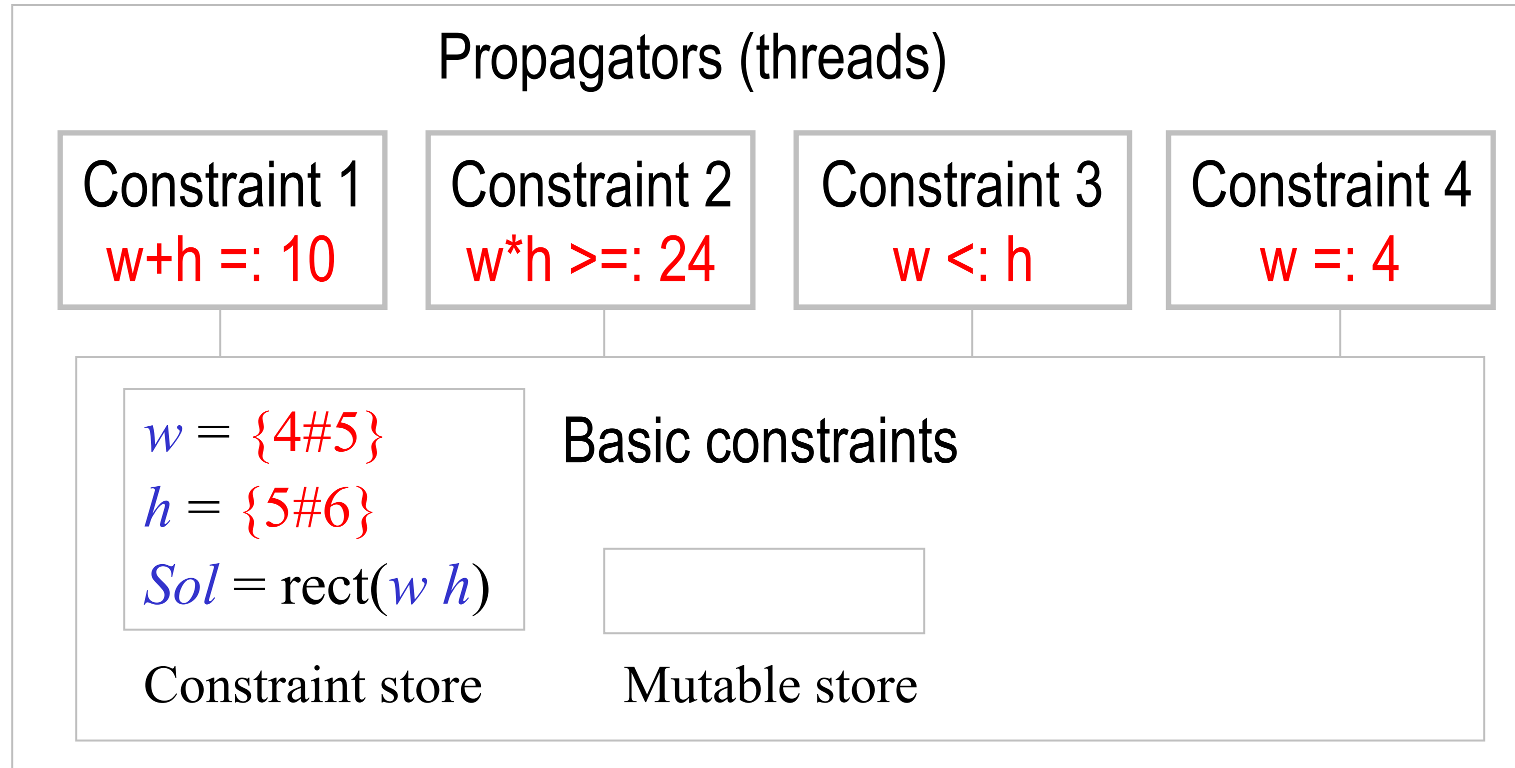
Divide original problem P into two new problems: (P ^ C) and (P ^ ¬C) and where C is a new constraint. The solution to P is the union of the solutions to the two new sub-problems.

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
```

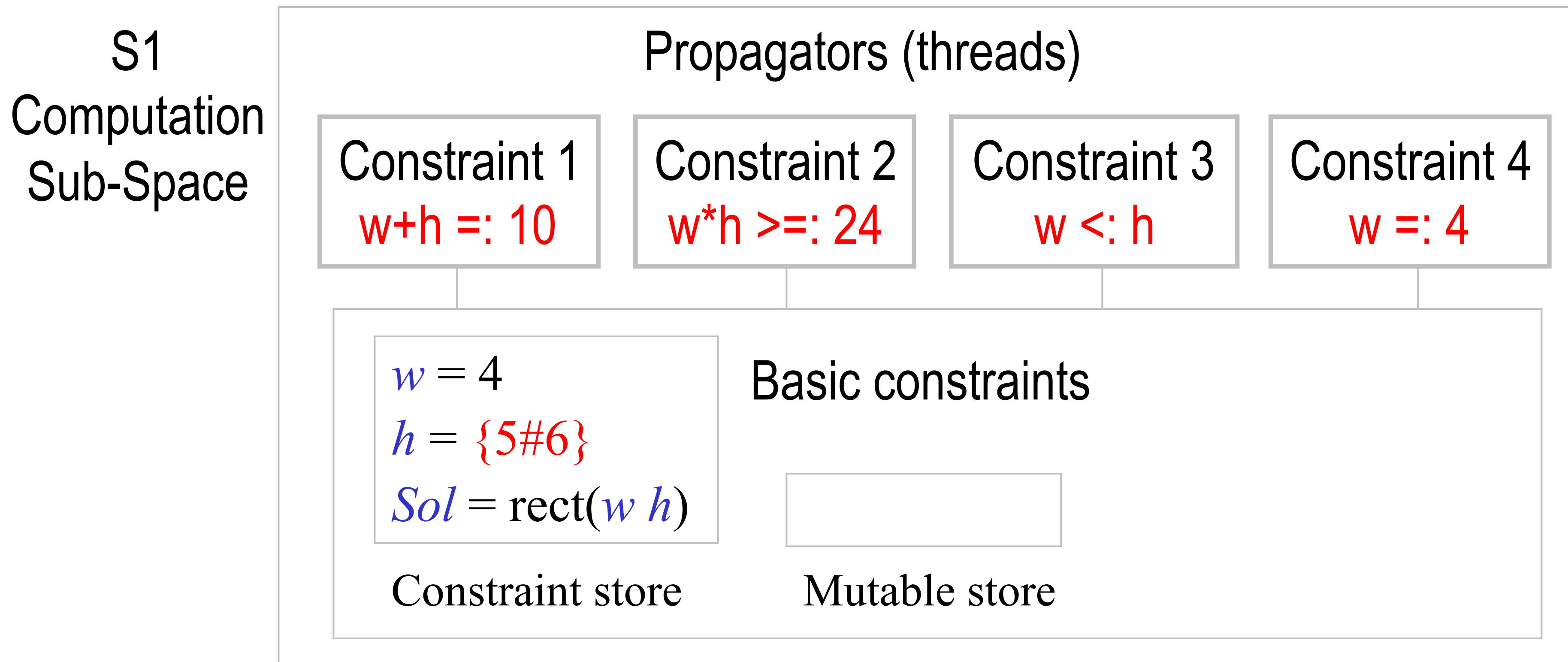
Computation Space Search: Rectangle example with w=4

S1
Computation
Sub-Space



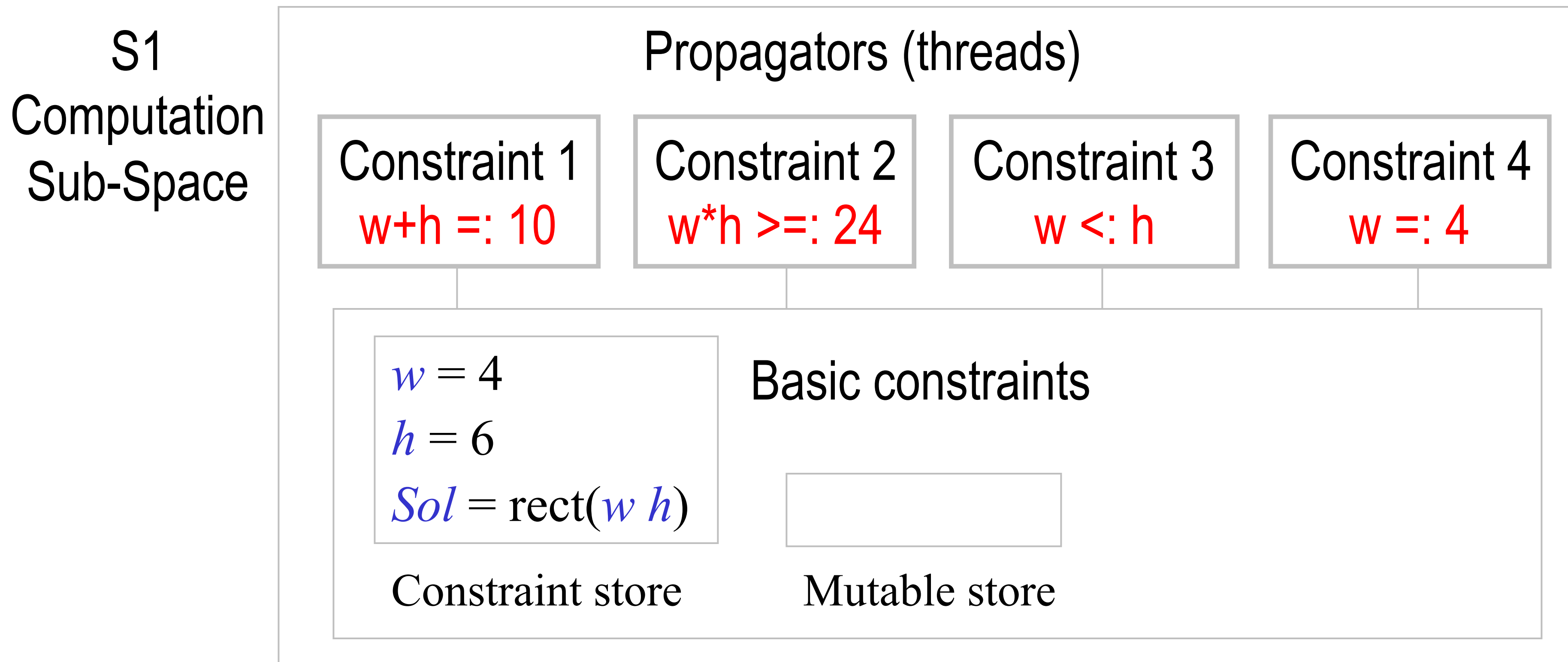
Constraint 4 implies that w = 4.

Computation Space Search: Rectangle example with w=4



Constraint 1 or 2 implies that h = 6.

Computation Space Search: Rectangle example with w=4



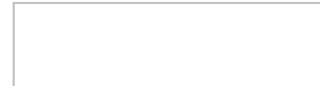
Since all the propagators are entailed by the store, their threads can terminate.

Computation Space Search: Rectangle example with w=4

S1
Computation
Sub-Space

$w = 4$
$h = 6$
Sol = rect(w h)

Basic constraints

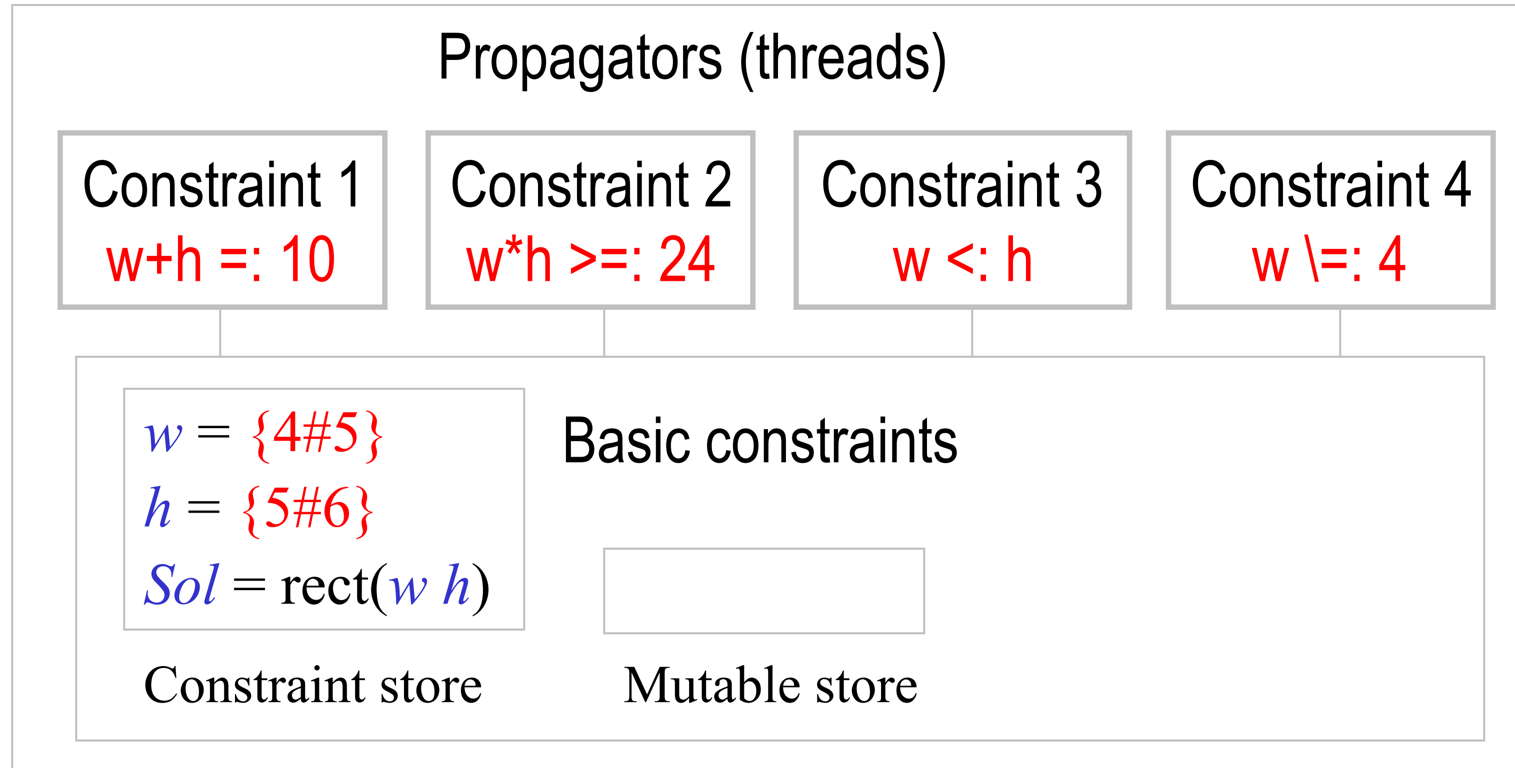


Constraint store

Mutable store

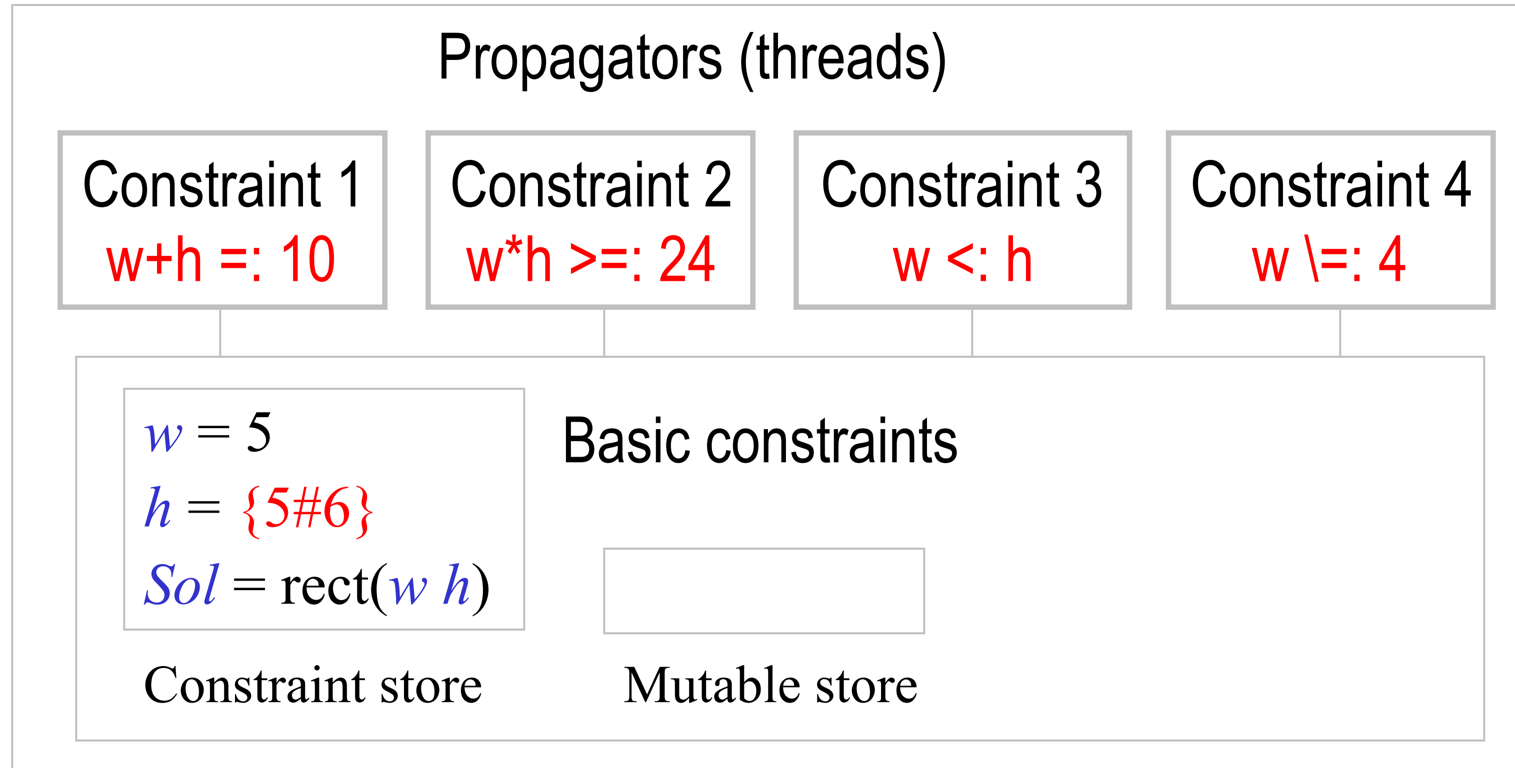
This is the final value store. A solution has been found.
The sub-space can now be **merged** with its parent computation space.

Computation Space Search: Rectangle example with w\=4



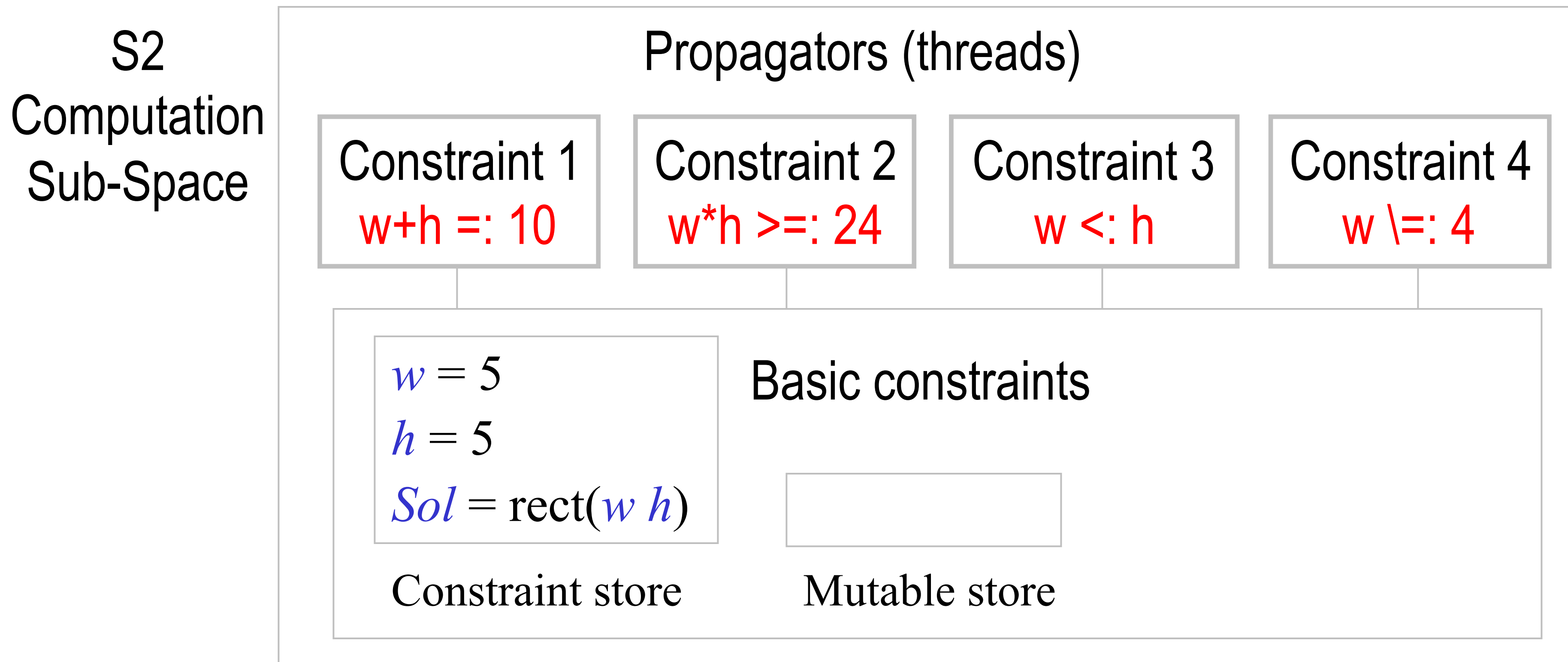
Constraint 4 implies that w = 5.

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**.

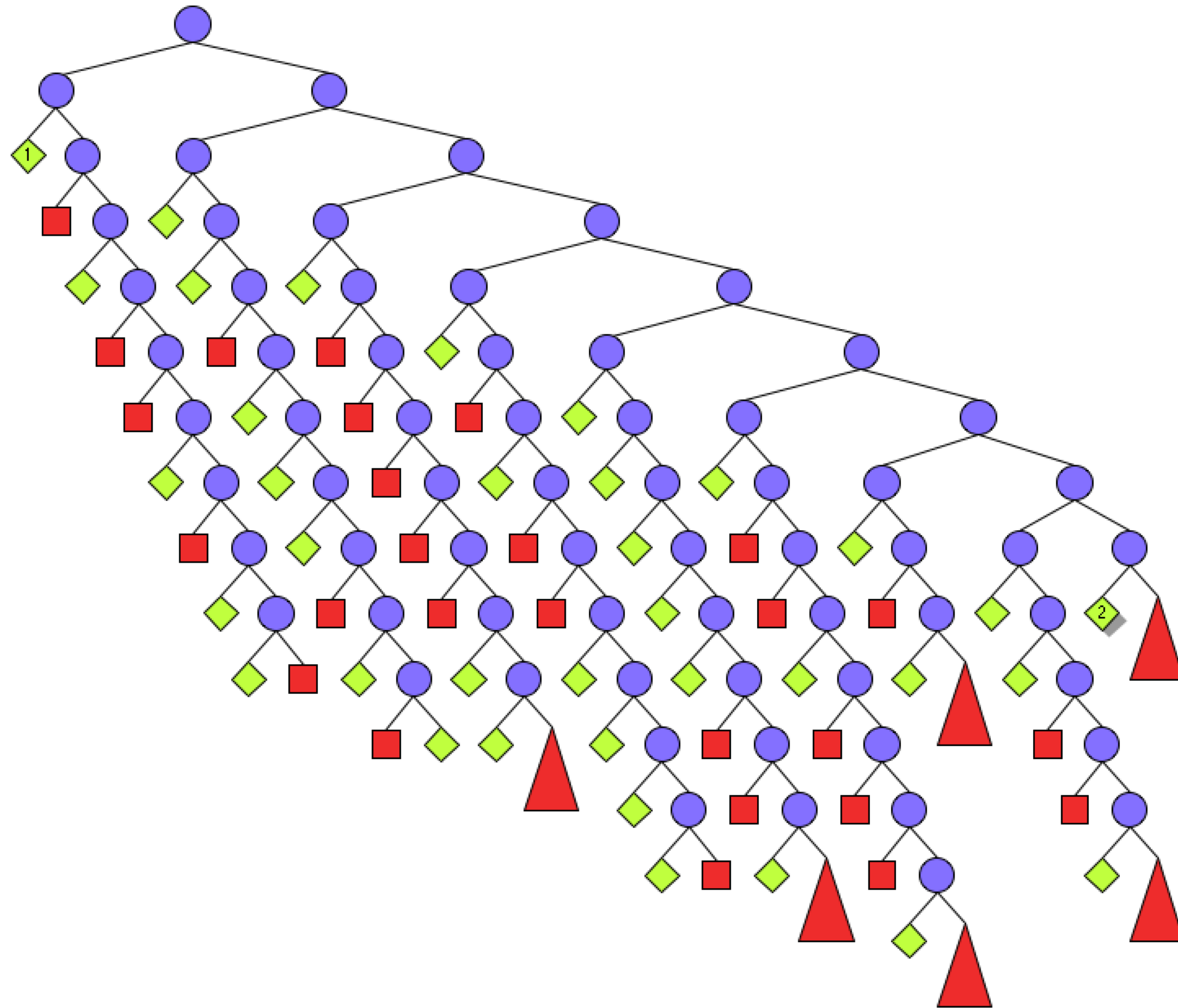
Finding palindromes (revisited)

- 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 ⟨s1⟩ [] … [] ⟨sn⟩ end
```

is a linguistic abstraction translated to:

```
case {Choose N}
of 1 then ⟨s1⟩
[] 2 then ⟨s2⟩
…
[] N then ⟨sn⟩
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
```