

# Spacetime programming

## A Synchronous Language for Constraint Search

Soutenance de thèse

**Pierre Talbot**  
(talbot@ircam.fr)

dirigée par Carlos Agon et encadrée par Philippe Esling

Institut de Recherche et Coordination Acoustique/Musique (IRCAM)  
Sorbonne Université

6 juin 2018



# Programmation par contraintes

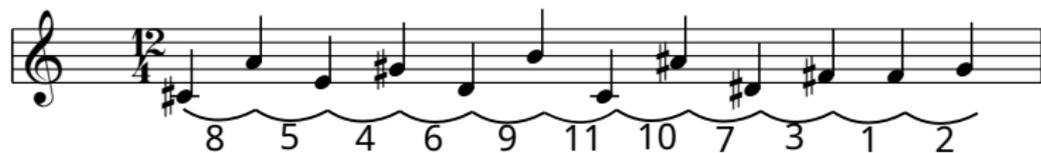
## “Holy grail of computing”

- ▶ Paradigme déclaratif pour résoudre des problèmes combinatoires.
- ▶ On déclare le problème et l'ordinateur le résout automatiquement pour nous.



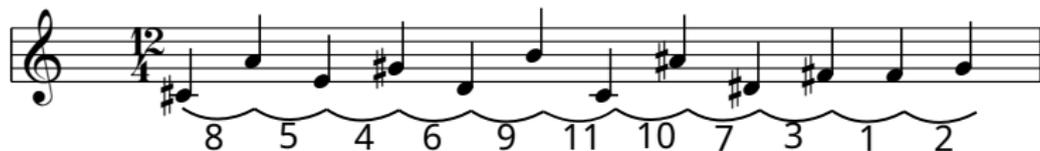
## Un exemple de problème de contraintes

Pour une série de 12 notes, chaque note et tout intervalle entre deux notes successives doit être distincts.



## Un exemple de problème de contraintes

Pour une série de 12 notes, chaque note et tout intervalle entre deux notes successives doit être distincts.



- ▶ On donne juste les relations entre les données qui nous intéressent.
- ▶ Mais on ne dit pas **comment** on arrive à la solution.

# Modélisation des séries tous intervalles

Pour une série de 12 notes, chaque note et tout intervalle entre deux notes successives doit être distincts.



Modèle en MiniZinc :

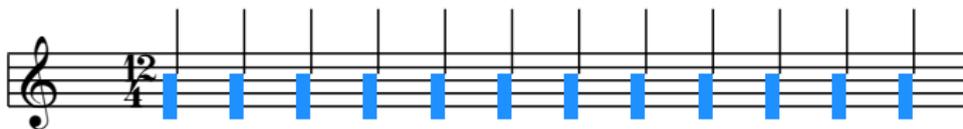
```
int : n = 12;
array[1..n] of var 1..n : pitches;
array[1..n-1] of var 1..n - 1 : intervals;
constraint forall(i in 1..n - 1)
    ( intervals [i] = abs(pitches[i+1] - pitches[i]));
constraint alldifferent ( pitches );
constraint alldifferent ( intervals );

solve satisfy;
```

# Comment marche un solveur de contraintes ?

## Nature NP-complete

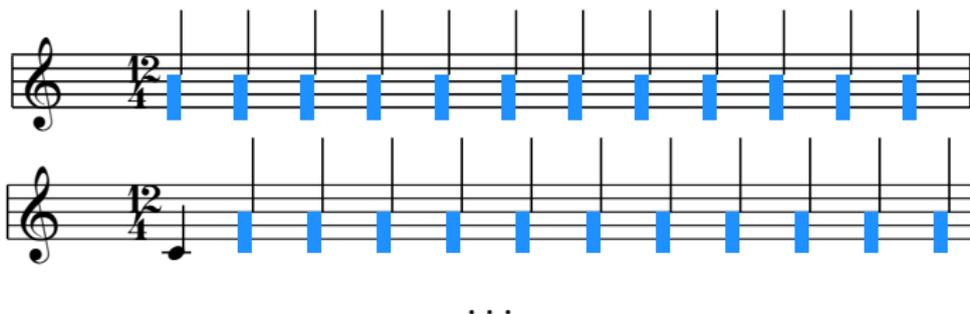
- ▶ Essayer toutes les combinaisons jusqu'à ce qu'on trouve une solution.
- ▶ Algorithme de *backtracking* construisant un arbre d'exploration.



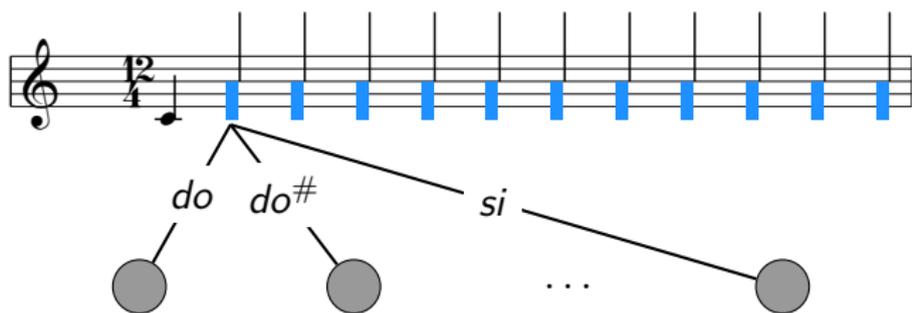
# Comment marche un solveur de contraintes ?

## Nature NP-complete

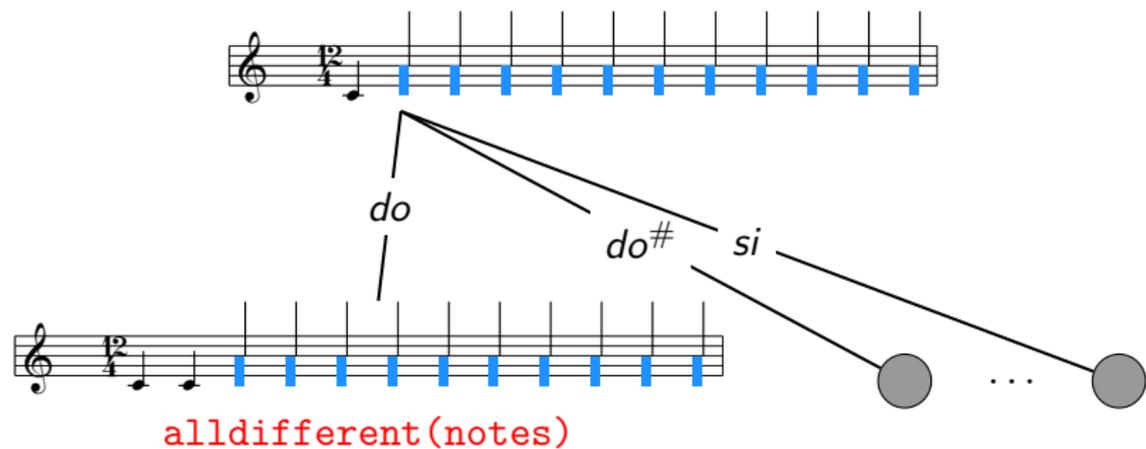
- ▶ Essayer toutes les combinaisons jusqu'à ce qu'on trouve une solution.
- ▶ Algorithme de *backtracking* construisant un arbre d'exploration.



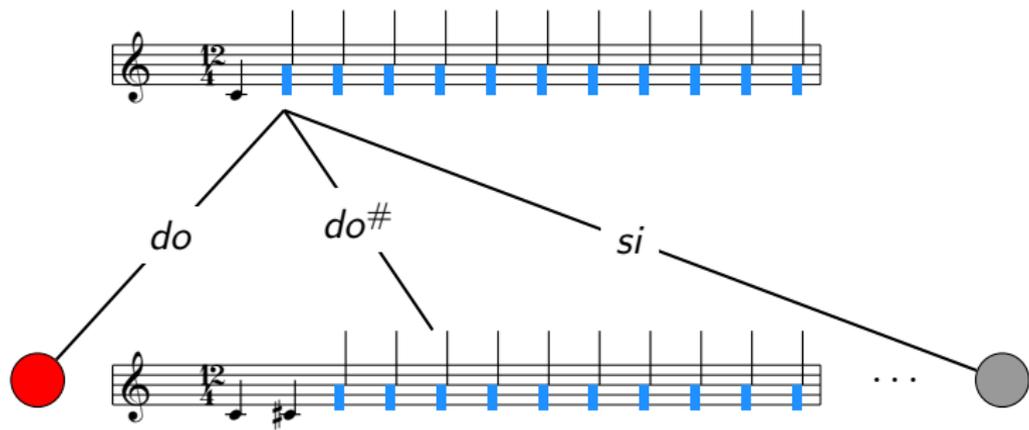
## Arbre d'exploration (étape 1)



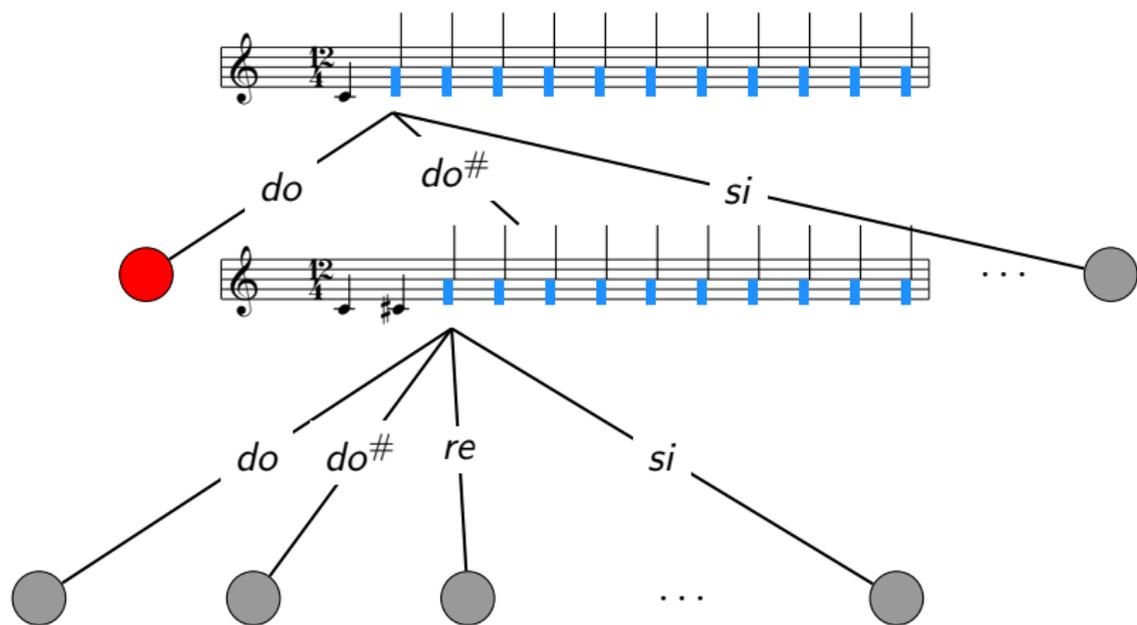
## Arbre d'exploration (étape 2)



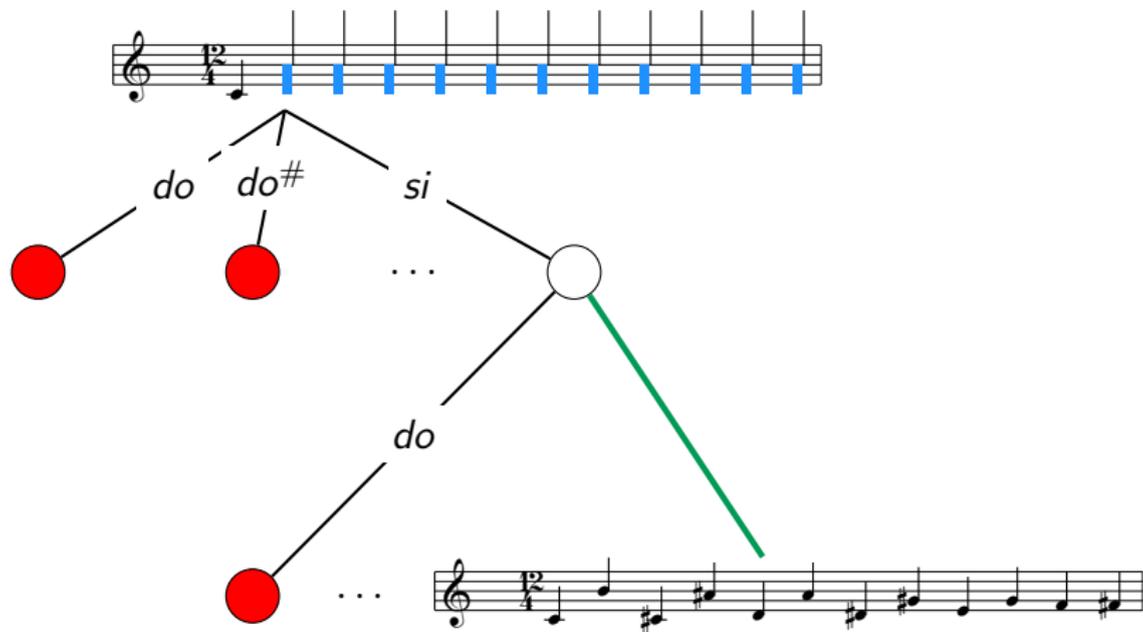
## Arbre d'exploration (étape 3)



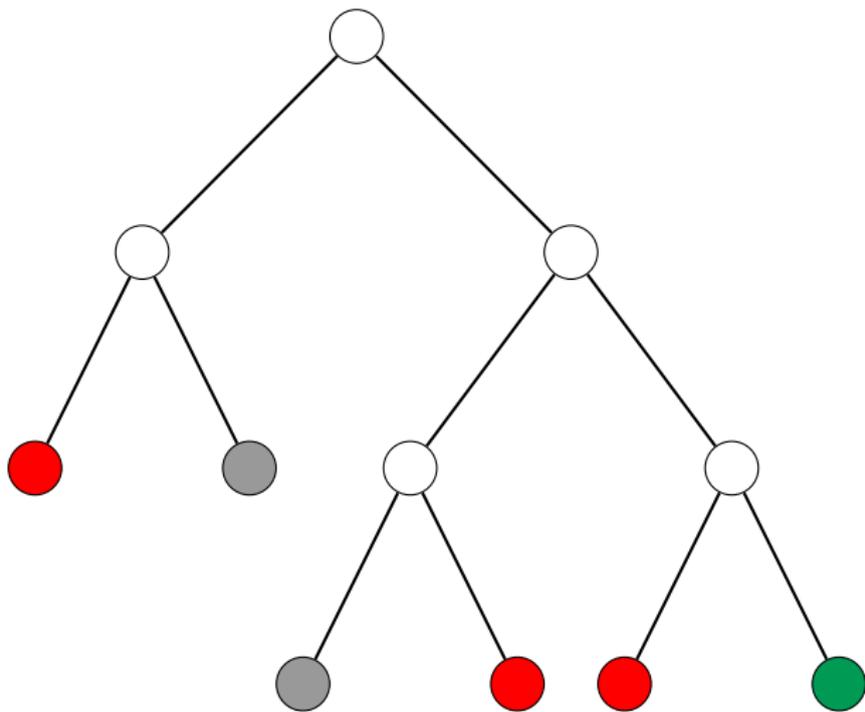
## Arbre d'exploration (étape 4)



# Arbre d'exploration (étape 5)



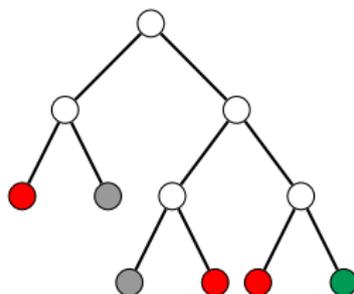
## Représentation abstraite d'un arbre d'exploration



# Problématique de ma thèse

## Holy grail ?

- ▶ L'arbre d'exploration est souvent trop grand pour trouver une solution en un temps raisonnable.
- ▶ **Stratégies d'exploration indispensables** afin d'être efficace.
- ▶ **Les stratégies sont dépendantes du problème : évaluation empirique.**



## Deux approches existantes

1. **Langages** (Prolog, MiniZinc,...) : Représentation lisible et compacte d'un problème mais choix de stratégies pré-définies limité.
  2. **Bibliothèques** (Choco, GeCode,...) : Hautement paramétrables et efficaces mais les solveurs sont très complexes, donc difficiles à étendre.
- ▶ En plus, composer des stratégies est impossible ou difficile dans les deux cas.

**On manque d'abstractions** pour définir, composer et étendre des stratégies d'exploration.

Proposition : **spacetime programming**.

SP = programmation par contraintes + programmation synchrone.

### Aspects synchrones idéaux pour composer des stratégies

- ▶ **Stratégie = processus** qui explore un arbre. On compose deux stratégies comme on compose deux processus.
- ▶ **Notion de temps logique** pour synchroniser les différentes stratégies sur l'exploration de l'arbre.

# Plan

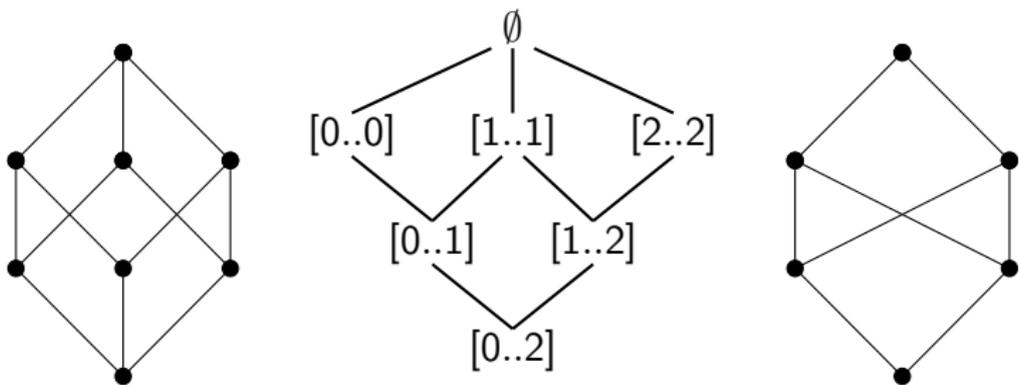
- ▶ Hiérarchie de treillis pour la résolution de contraintes
- ▶ Spacetime programming
  - ▶ Modèle d'exécution et syntaxe
  - ▶ Exemple de composition de stratégies
  - ▶ Notion d'univers
- ▶ Conclusion

# Quelques définitions

## Treillis

Soit un ordre partiel  $\langle L, \leq \rangle$ ,  $L$  est un treillis si :

- ▶ Pour tout élément  $x, y \in L$ , il existe une borne inférieure et une borne supérieure de  $x$  et  $y$  dans  $L$ .
- ▶ Il est borné s'il y a une borne inférieure/supérieure de tous les éléments, notée  $\perp$  et  $\top$ .



## Vue algébrique des treillis

Un treillis peut être vu comme un système d'information et une structure algébrique  $\langle S, \models, \sqcup, \sqcap, \perp, \top \rangle$  où

- ▶  $x \models y$  est l'ordre voulant dire "on peut déduire  $y$  de  $x$ ".
- ▶  $x \sqcup y$  est appelé "join" et est la borne supérieure de  $x$  et  $y$ .
- ▶  $x \sqcap y$  est appelé "meet" et est la borne inférieure de  $x$  et  $y$ .
- ▶  $\perp$  représente l'absence d'information.
- ▶  $\top$  représente toute l'information.

### Exemples avec le treillis des intervalles

- ▶  $[0..0] \models [0..2]$
- ▶  $[0..0] \sqcup [1..1] = \top$
- ▶  $[1..1] \sqcap [2..2] = [1..2]$

# Hiérarchie de treillis

Les solveurs de contraintes travaillent sur une structure hiérarchique.

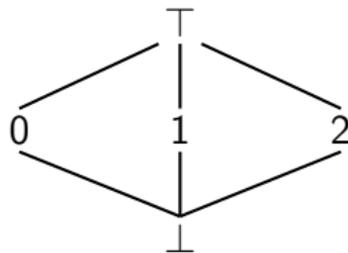
Niveau	Entité
$L_0$	Valeur
$L_1$	Domaine
$L_2$	Collection de variables
$L_3$	CSP
$L_4$	Arbre d'exploration
$L_5$	Collection d'arbres
$L_*$	Sélection d'algorithmes

Chaque niveau est un treillis  $L_i$  dérivant de l'ensemble des parties du treillis  $L_{i-1}$ .

# Valeur (L0)

## Treillis plat

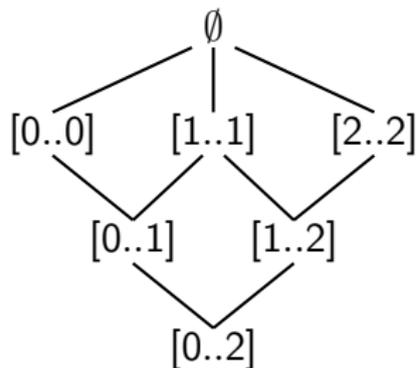
$$L_0 = \{\top\} \oplus \{0, 1, 2\} \oplus \{\perp\}$$



# Domaine (L1)

## Exemple de treillis $L_1$

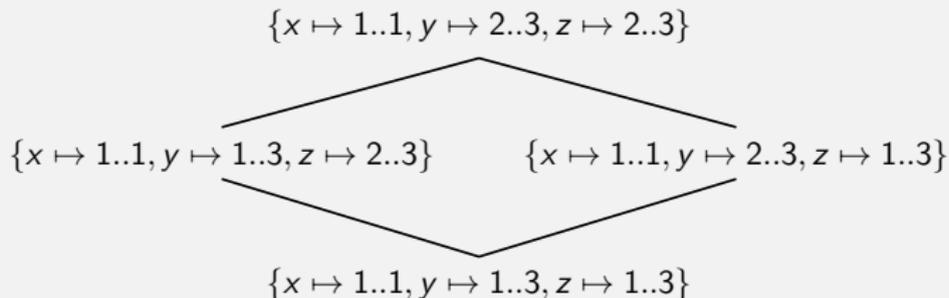
$$L_1 = \{T\} \cup \{(l, u) \in S \times S \mid l \leq_S u\}$$



# Collection de variables (L2)

## Treillis $L_2$

$$L_2 = \{S \in \mathcal{P}(\text{Loc} \times L_1) \mid \forall x, y \in S. \text{loc}(x) = \text{loc}(y) \Rightarrow x = y\}$$

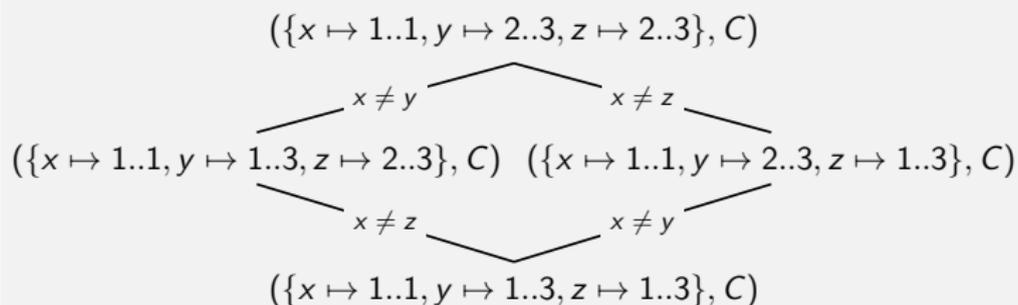


# Problème de satisfaction de contraintes (L3)

## Treillis $L_3$

$L_3 = L_2 \times \mathcal{P}(C)$  où  $C$  est l'ensemble de toutes les contraintes.

Exemple :  $\langle \{x \mapsto 1..1, y \mapsto 1..3, z \mapsto 1..3\}, \{x \neq y, x \neq z\} \rangle$

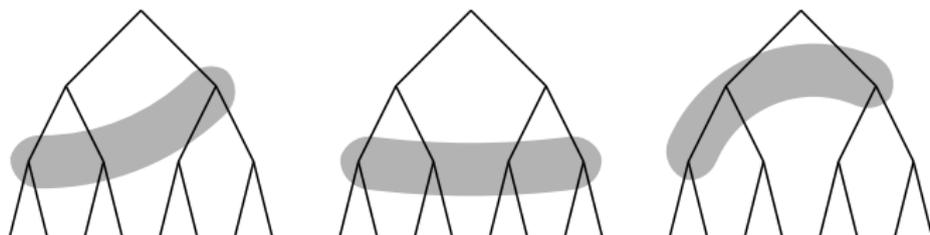


# Résolution : Arbre d'exploration ( $L_4$ )

## Treillis anti-chaine $L_4$

$$L_4 = \{Q \in \mathcal{P}(L_3) \mid \forall a, b \in Q. a \models_3 b \Rightarrow a = b\}$$

On représente un arbre dans  $L_4$  par ses nœuds à la frontière de l'arbre, on "oublie" ses nœuds intérieurs.



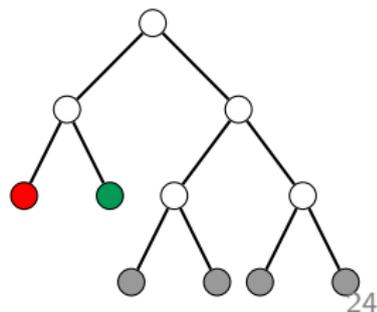
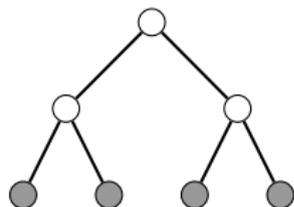
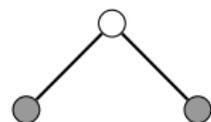
# Résolution : Collection d'arbres (L5)

## Treillis $L_5$

$$L_5 = \mathcal{P}(L_4)$$

## Iterative deepening search (IDS), Korf 85'

- ▶ Combine les avantages du DFS et de BFS.
- ▶ On explore d'abord l'arbre à profondeur fixée  $N$ , et puis on recommence avec  $N + 1$  jusqu'à ce qu'on trouve une solution.



# Hiérarchie de treillis

## Structure homogène comprenant modélisation et résolution.

- ▶ Dérivation successive d'un ensemble de base  $S$  (e.g.  $\mathbb{N}$ ) :
  - ▶  $L_0 = \{\top\} \oplus S \oplus \{\perp\}$ .
  - ▶ Intuitivement, on a (plus ou moins)  $L_i \subseteq \mathcal{P}(L_{i-1})$ .

Niveau	Entité	Exemple
$L_0$	Valeur	Variable logique
$L_1$	Domaine	Ensemble d'intervalles, "bit array"
$L_2$	Collection de variables	Tableau de variables
$L_3$	CSP	MiniZinc, CHR
$L_4$	Arbre d'exploration	Prolog, Oz
$L_5$	Collection d'arbres	"Search combinators"
$L_n$	Sélection d'algorithmes	EPS, sunny-cp2

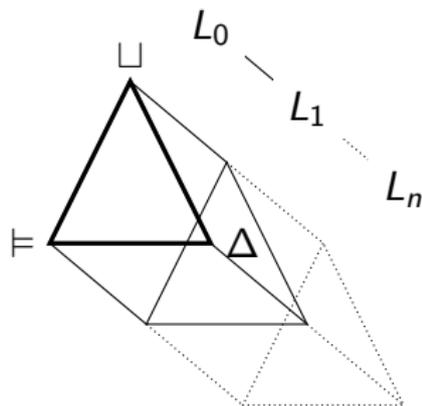
# Connexion algorithmes et langages

## Programmation concurrente par contraintes (CCP)

$\langle L_3, \models, \sqcup \rangle$  issu de CCP (Saraswat 90') où

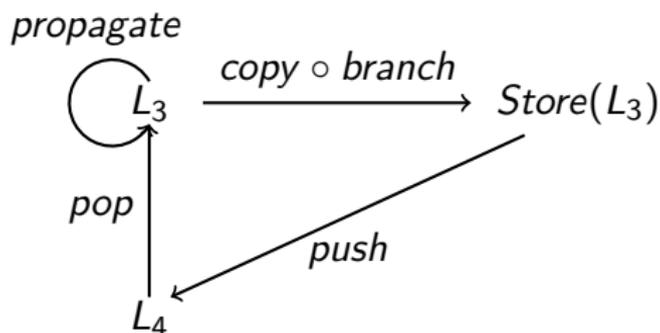
- ▶  $\models$  permet de demander de l'information (condition).
- ▶  $\sqcup$  permet d'ajouter de l'information.

On utilise l'opérateur  $\Delta$ , une variante de  $\sqcup$ , pour *backtracker*.  
Trois opérateurs pour chaque niveau de la hiérarchie.



# Stratégie d'empilage

Algorithme standard de résolution de contraintes :



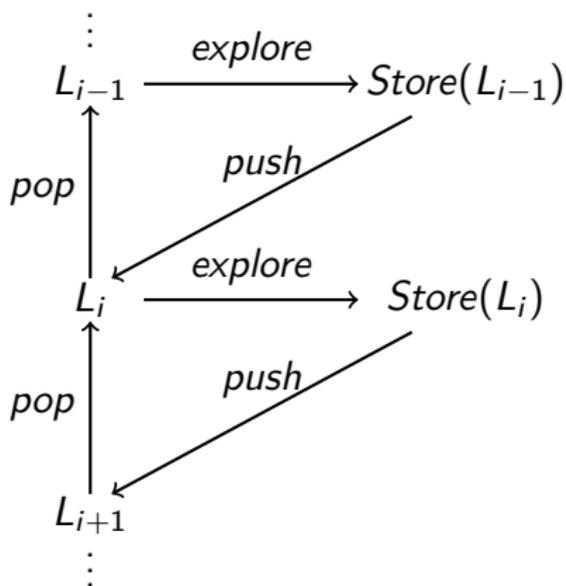
## Traverser les niveaux de la hiérarchie

Une stratégie d'empilage est une paire  $(push, pop)$  telle que :

$$\begin{array}{ll} push : L_i \times \mathcal{P}(L_{i-1}) \rightarrow L_i & pop : L_i \rightarrow L_i \times L_{i-1} \\ push(q, s) \mapsto q \sqcup_i s & pop(q) \mapsto (q \Delta_i \{v\}, v) \end{array}$$

## Stratégie d'exploration abstraite

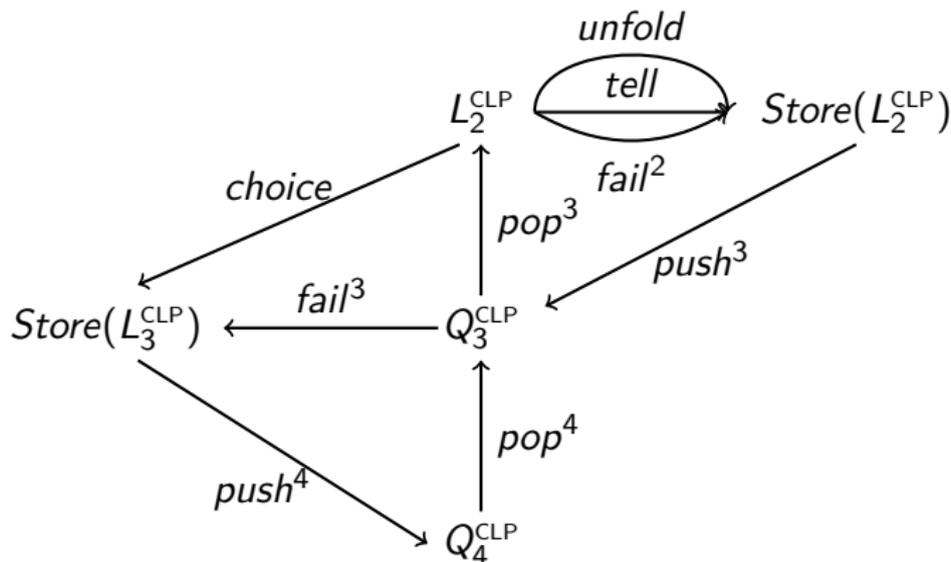
En utilisant plusieurs stratégies d'empilage, on peut définir un **algorithme** sur plusieurs niveaux.



## Sémantique basé sur la hiérarchie de treillis

La hiérarchie permet également de définir la **sémantique** des langages par contraintes.

Sémantique de la programmation logique par contraintes sous forme de diagramme :



# Résumé

- ▶ Proposition d'une formalisation hiérarchique des solveurs de contraintes.
- ▶ Connexion entre algorithmes et sémantique des langages de programmation par contraintes avec les opérateurs algébriques.

*Spacetime programming* est basé sur cette hiérarchie.

# Plan

- ▶ Hiérarchie de treillis pour la résolution de contraintes
- ▶ Spacetime programming
  - ▶ Modèle d'exécution et syntaxe
  - ▶ Exemple de composition de stratégies
  - ▶ Notion d'univers
- ▶ Conclusion

# Spacetime programming

- ▶ **Objectif pratique** : Un langage pour définir et composer des stratégies d'exploration.
- ▶ **Objectif langage** : Explorer un paradigme qui combine les contraintes et la programmation synchrone.

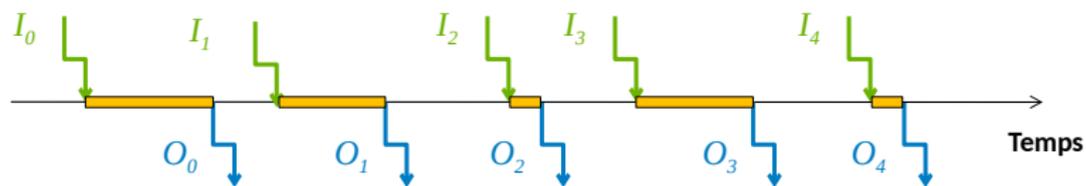
## Apports d'un nouveau paradigme ?

- ▶ Étude du temps logique comme outil de synchronisation des stratégies d'exploration.
- ▶ Étude d'un cadre commun aux langages avec contraintes.

# Paradigme synchrone

- ▶ Inventé dans les années 80' pour gérer les systèmes qui :
  - ▶ Réagissent à de nombreuses entrées simultanées.
  - ▶ Sont en interaction continue avec l'environnement.

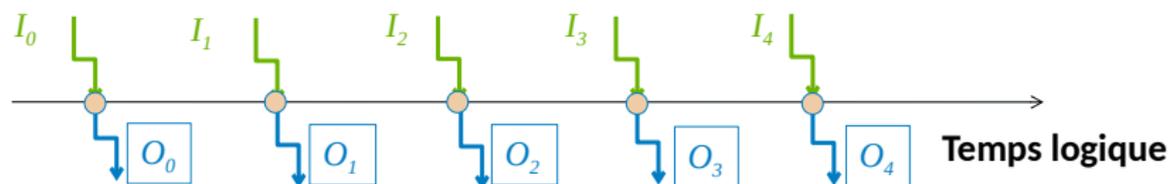
Division de l'exécution en instants logiques :



# Paradigme synchrone

- ▶ Inventé dans les années 80' pour gérer les systèmes qui :
  - ▶ Réagissent à de nombreuses entrées simultanées.
  - ▶ Sont en interaction continue avec l'environnement.

**Hypothèse synchrone** : Un instant ne prend pas de temps :



- ▶ Forte garantie de **déterminisme** malgré la notion de **concurrence**.
- ▶ Une entrée n'a qu'une sortie possible.

## Un langage synchrone : Esterel (*Berry et al.*, 92')

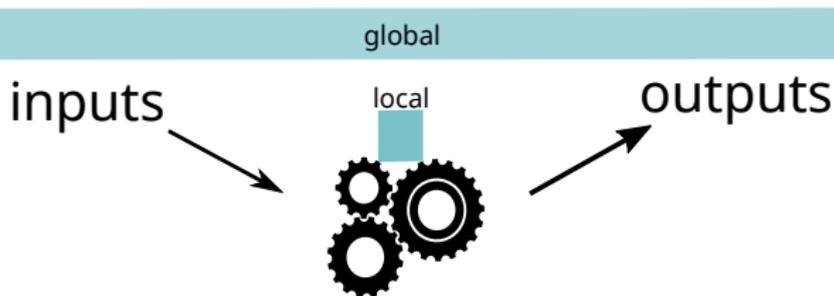
- ▶ “ABRO est le Fibonacci de Esterel”.
- ▶ Variante ABO : Émet  $O$  dès que  $A$  et  $B$  sont arrivés, en plus on compte les émissions de  $O$ .

```
module ABO :  
  input A, B;  
  output O := 0 : integer;  
  loop  
    [ await A || await B ];  
    emit O(pre(?O) + 1);  
    pause;  
  end loop  
end module
```

# Modèle d'exécution synchrone

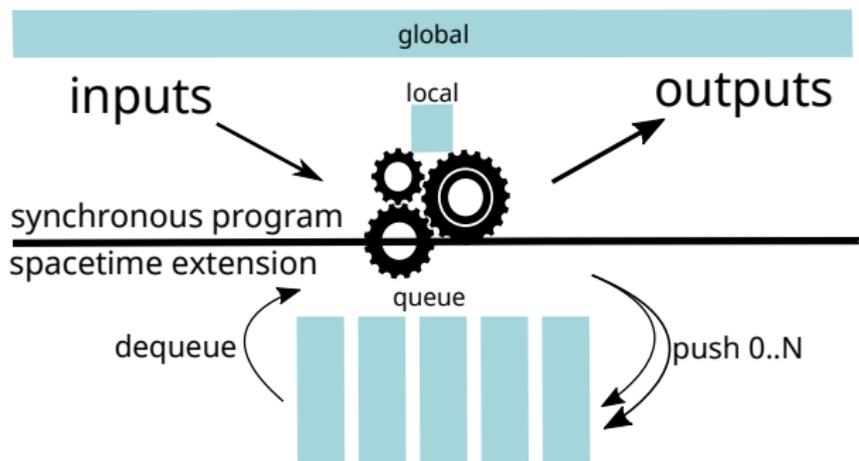
## Temps logique

- ▶ Notion de temps logique pour coordonner les processus.
- ▶ À chaque **instant**, tous les processus sont exécutés simultanément et puis s'attendent jusqu'à ce qu'ils atteignent tous une "barrière" (une unité de temps s'est écoulée).



## Modèle d'exécution de *spacetime*

- ▶ L'arbre d'exploration est représenté sous forme de file de nœuds.
- ▶ On extrait **un nœud par instant** qui est donné en entrée au programme.
- ▶ À chaque instant, de nouveaux nœuds sont ajoutés dans la file.



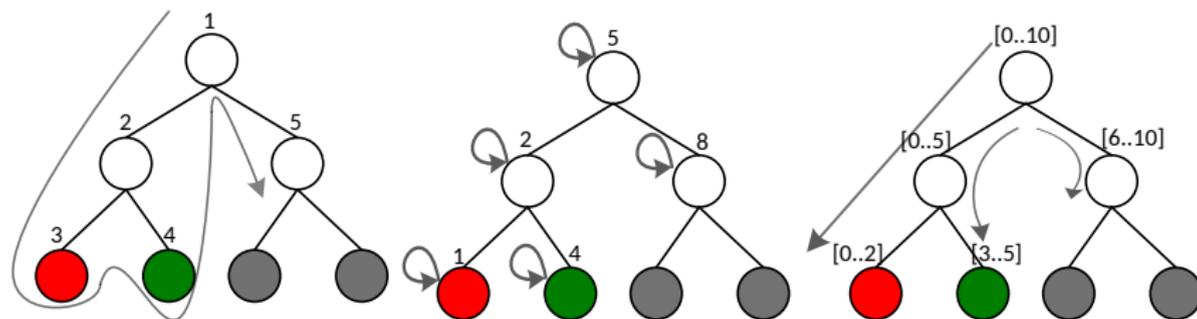
# Attributs *spacetime*

## Problème

Comment différencier les variables dans l'état interne/global et celles dans la file ?

Utilisation d'un attribut pour situer une variable dans l'espace et le temps.

- ▶ `single_space` : variable **globale** à l'arbre d'exploration.
- ▶ `single_time` : variable **locale** à un instant.
- ▶ `world_line` : variable **backtrackable** dans la file de nœuds.



# Spacetime programming : Syntaxe

$\langle p, q, \dots \rangle ::=$	<b>fragment CCP (<math>L_3</math> et en-dessous)</b>
<i>spacetime</i> Type $x = e$	(déclaration de variable)
<b>when</b> $x \mid= y$ <b>then</b> $p$ <b>else</b> $q$ <b>end</b>	(ask)
$x \leftarrow e$	(tell)
$f(\text{args})$	(appel de fonction)
	<b>fragment synchrone</b>
<b>par</b> $p \parallel q$ <b>end</b>	(composition parallèle disjonctive)
<b>par</b> $p \langle \rangle q$ <b>end</b>	(composition parallèle conjonctive)
$p ; q$	(composition séquentielle)
<b>loop</b> $p$ <b>end</b>	(boucle infinie)
<b>pause</b>	(délai)
	<b>fragment arbre d'exploration (<math>L_4</math>)</b>
<b>space</b> $p$ <b>end</b>	(création d'une branche)
<b>prune</b>	(coupage d'une branche)
	<b>fragment des univers (<math>L_5</math> et au-dessus)</b>
<b>universe with</b> $q$ <b>in</b> $p$ <b>end</b>	(création d'un univers)

# Plan

- ▶ Hiérarchie de treillis pour la résolution de contraintes
- ▶ Spacetime programming
  - ▶ Modèle d'exécution et syntaxe
  - ▶ Exemple de composition de stratégies
  - ▶ Notion d'univers
- ▶ Conclusion

## Composition de stratégies

Chaque processus génère un arbre qu'on peut combiner de manière différente.

- ▶  $p ; q$  concatène les branches de  $p$  et  $q$ .
- ▶  $p \parallel q$  fait l'union des branches.
- ▶  $p \langle \rangle q$  fait l'intersection des branches.

Pour illustrer ces opérateurs, on crée différentes sous-stratégies qu'on assemble ensuite :

```
par base_tree() <> propagate() <> bound_depth(2) end
```

# Composition séquentielle

Création de l'espace d'exploration :

```
class Solver {  
  world_line VStore domains;  
  world_line CStore constraints;  
  public Solver(VStore domains,  
               CStore constraints) {  
    this.domains = domains;  
    this.constraints = constraints;  
  }  
  
  proc base_tree =  
    loop  
      single_time IntVar x = inputOrder(domains);  
      single_time Integer v = middleValue(x);  
      space constraints <- x.le(v) end;  
      space constraints <- x.gt(v) end;  
      pause;  
    end
```

} Membres avec l'attribut *spacetime*

} Constructeur Java

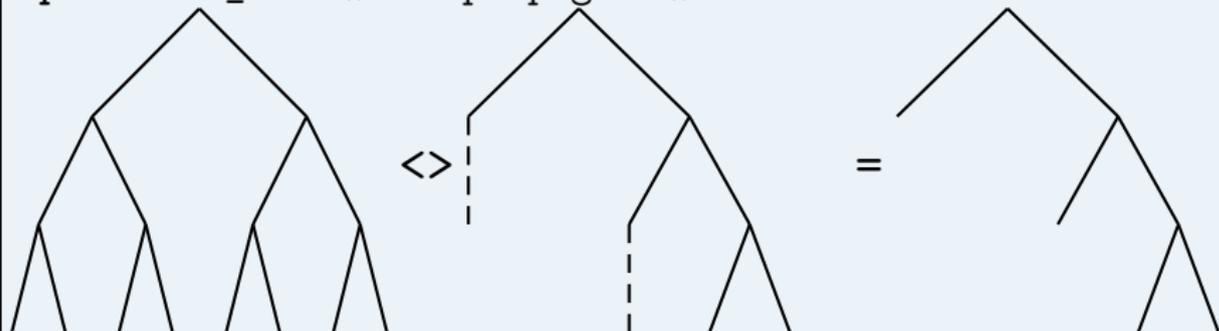
} Stratégie de branchement

} Création des branches

# Propagation

```
proc propagate =  
  loop  
    single_time ES consistency <- read constraints.propagate(readwrite domains);  
    when consistency != unknown then  
      prune;  
    end  
    pause;  
  end  
end
```

```
par base_tree() <> propagate() end
```

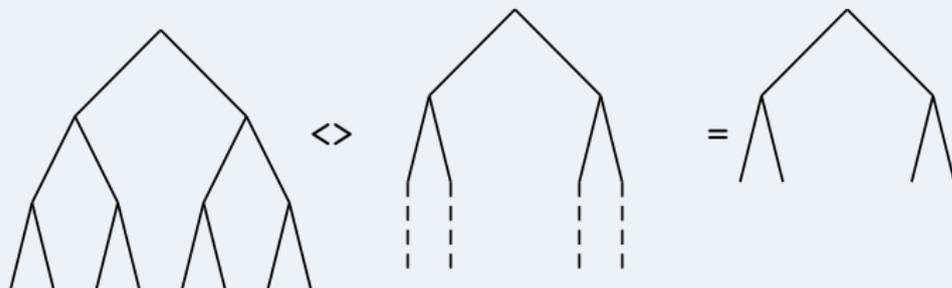


# Limiter l'exploration en profondeur

```
proc bound_depth(limit) =  
  world_line LMax depth = new LMax(0);  
  loop  
    when depth != limit then  
      prune;  
    end  
    pause;  
    readwrite depth.inc();  
  end  
end
```

} Coupe les branches quand on atteint la limite

```
par base_tree <> bound_depth(2) end
```

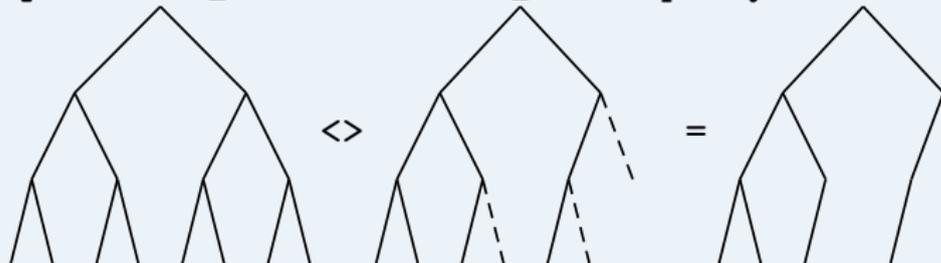


## Limiter l'exploration par les divergences

```
proc bound_discrepancy(limit) =  
  world_line LMax value = new LMax(0);  
  loop  
    space nothing end;  
    when value |= limit then  
      readwrite value.inc();  
      prune  
    end  
    pause;  
  end  
}
```

} Branche gauche  
} Branche droite

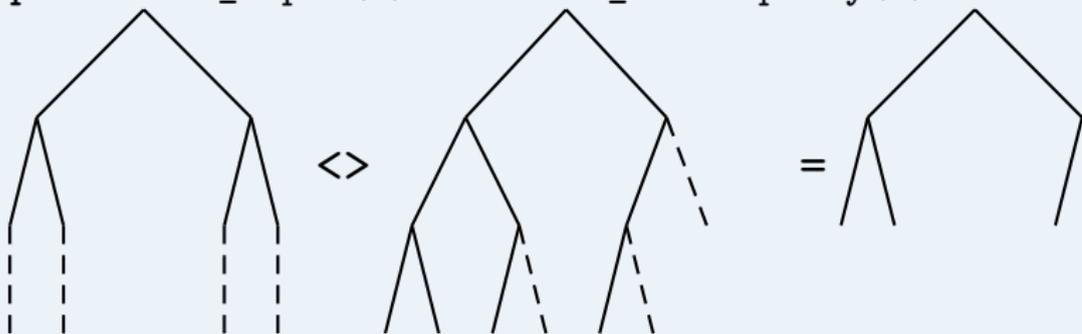
```
par base_tree <> bound_discrepancy(1) end
```



## Composition d'arbres par intersection

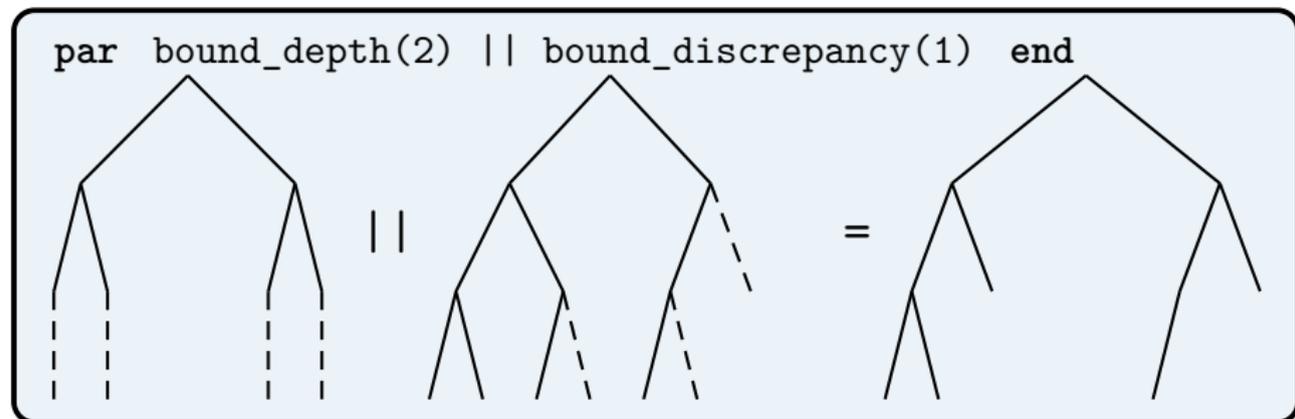
On peut composer la limite de profondeur et de divergence par intersection :

```
par bound_depth(2) <> bound_discrepancy(1) end
```



## Composition d'arbres par union

On peut composer la limite de profondeur et de divergence par union :



```
par  
<> base_tree()  
<> propagate()  
<> par bound_depth(2) || bound_discrepancy(1) end  
end
```

- ▶ **Communication** entre les stratégies avec les variables domains et constraints.
- ▶ **Compositionnalité et ré-utilisabilité** : chaque stratégie est écrite indépendamment.

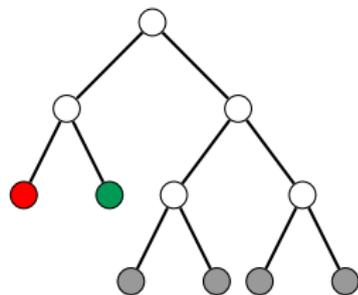
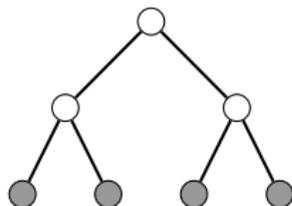
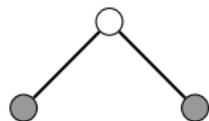
# Plan

- ▶ Hiérarchie de treillis pour la résolution de contraintes
- ▶ Spacetime programming
  - ▶ Modèle d'exécution et syntaxe
  - ▶ Exemple de composition de stratégies
  - ▶ Notion d'univers
- ▶ Conclusion

# Stratégie sur $L_5$

## Vers des stratégies sur $L_5$

- ▶ Pour certaines stratégies, on a besoin de contrôler la file de nœuds du programme.
- ▶ Les fonctions *pop* et *push* sont appelées implicitement sur  $L_3$  et  $L_4$  dans le modèle actuel.



## Le concept d'univers

- ▶ Notion d'univers encapsulant la file de nœuds.
- ▶ Au lieu d'explorer un nœud par instant, on explore *un arbre par instant* en augmentant la limite  $N$ .

### Inspirations : Raffinement spatial et temporel

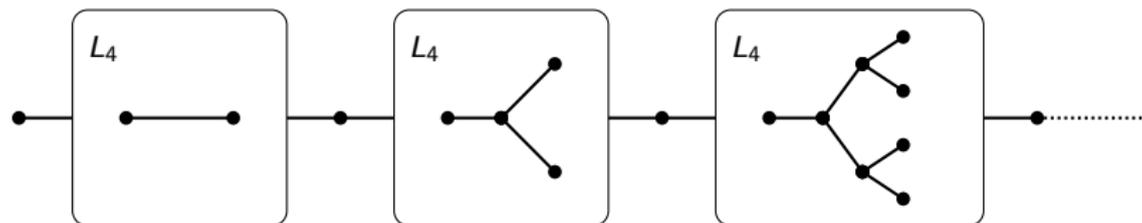
- ▶ Raffinement temporel (Pasteur et al. 13', Gemünde et al. 13') : "Échelle de temps" où les processus sont exécutés plus vite.
- ▶ "Computation space" (Schulte 00') : Extension de Oz (Van Roy et al. 04') où l'exploration est encapsulée dans un espace.

# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit .inc();  
  end  
end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.

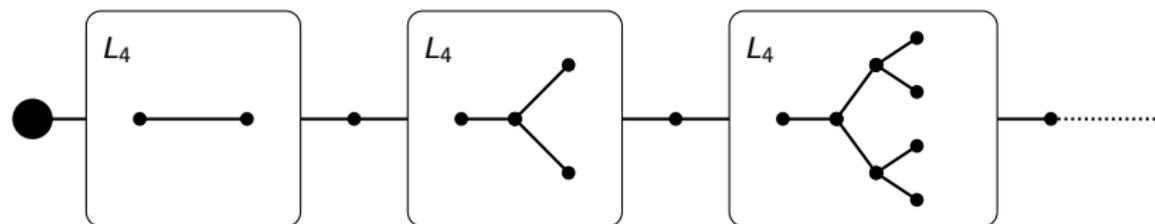


# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit .inc();  
  end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.

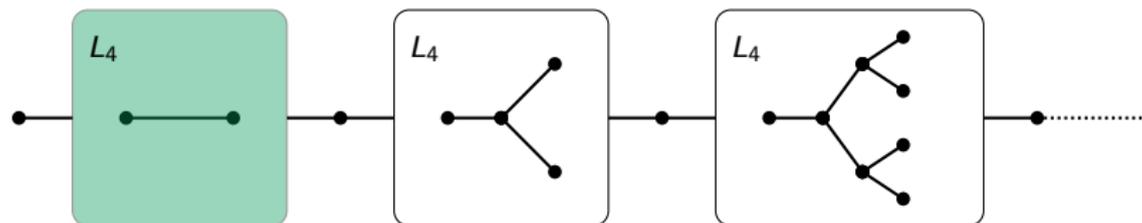


# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit .inc();  
  end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.

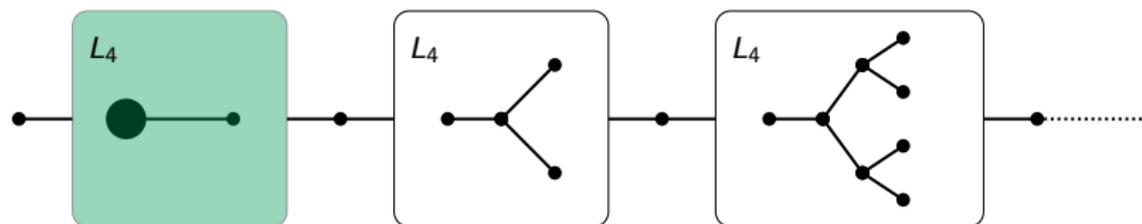


# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit.inc();  
  end  
end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.

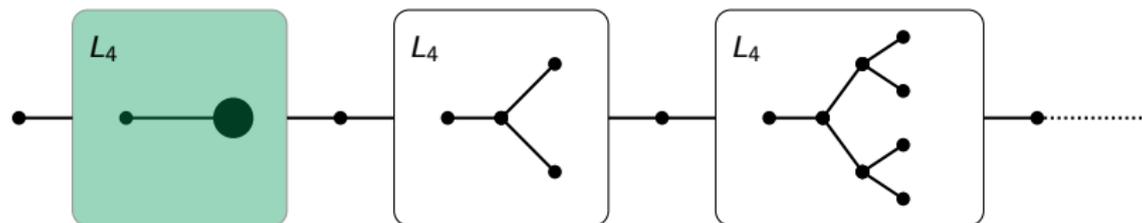


# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit .inc();  
  end  
end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.

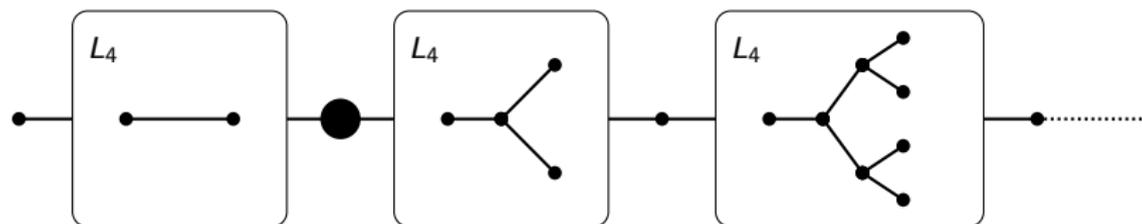


# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit .inc();  
  end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.

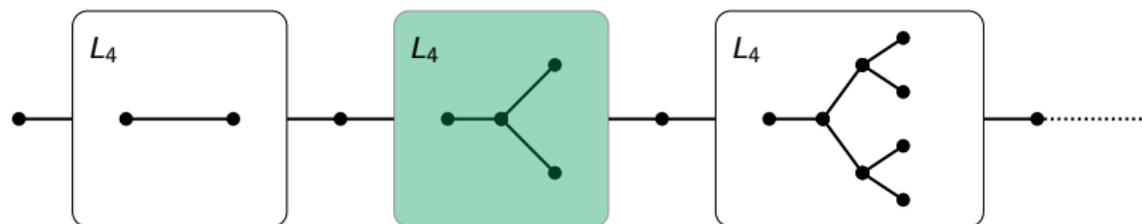


# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit.inc();  
  end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.

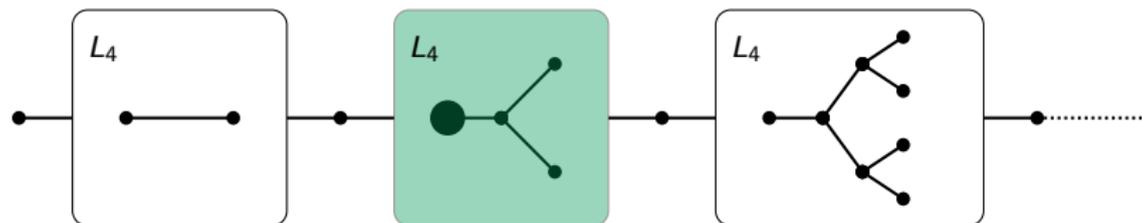


# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit.inc();  
  end  
end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.

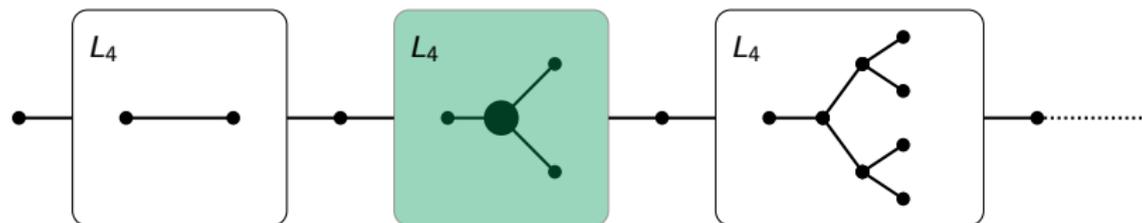


# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit.inc();  
  end  
end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.

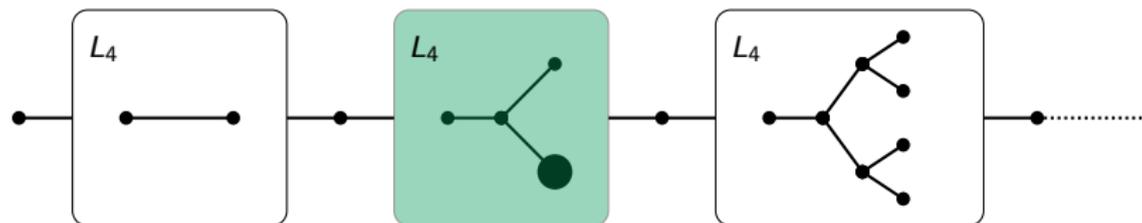


# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit.inc();  
  end  
end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.

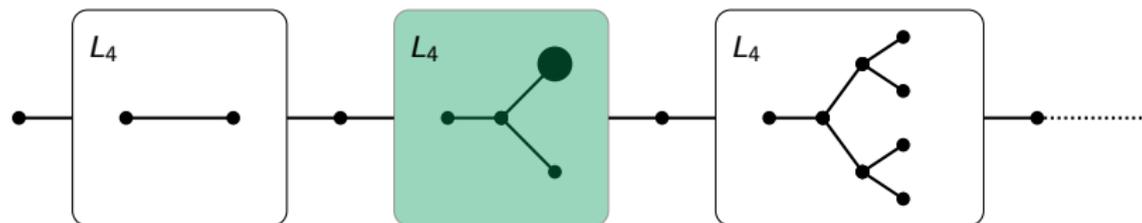


# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit.inc();  
  end  
end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.

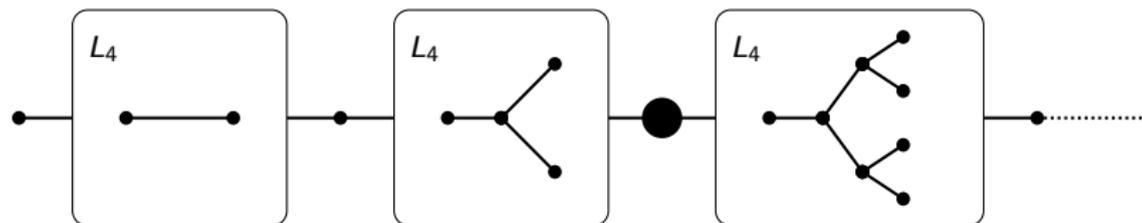


# Exploration en profondeur itérative

On encapsule dans un univers l'exploration limitant la profondeur :

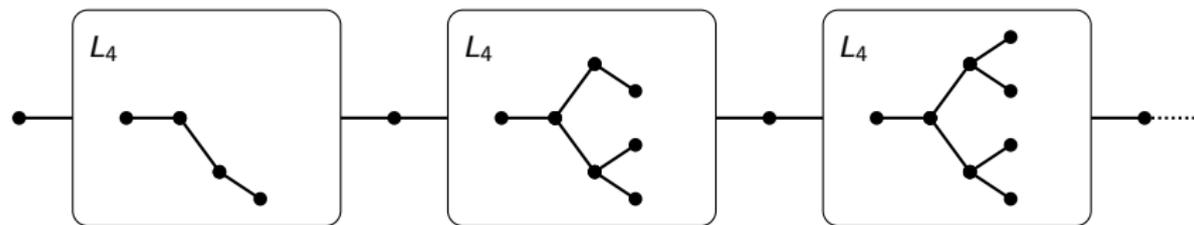
```
proc ids =  
  single_space LMax limit = new LMax(0);  
  single_time StackLR q = new StackLR();  
  loop  
    universe with q in  
      bound_depth(limit);  
    end  
    pause;  
    limit .inc();  
  end  
end
```

La file de nœud est un treillis déclaré comme une variable dans *spacetime*.



## Une deuxième stratégie par redémarrage

- ▶ Limited discrepancy search (LDS), (Harvey and Ginsberg 95').
- ▶ Code similaire à IDS mais avec le processus `bound_discrepancy`.



## Composition des univers

- ▶ Dans un instant, les univers sont exécutés en mode synchrone.
- ▶ La sémantique de composition s'étend aux univers automatiquement.

par ids() <> lds() end

Itération 1



## Composition des univers

- ▶ Dans un instant, les univers sont exécutés en mode synchrone.
- ▶ La sémantique de composition s'étend aux univers automatiquement.

par `ids()` <> `lds()` end

Itération 2



## Composition des univers

- ▶ Dans un instant, les univers sont exécutés en mode synchrone.
- ▶ La sémantique de composition s'étend aux univers automatiquement.

par `ids()`  $\langle \rangle$  `lds()` end

Itération 3



# Plan

- ▶ Hiérarchie de treillis pour la résolution de contraintes
- ▶ Spacetime programming
  - ▶ Modèle d'exécution et syntaxe
  - ▶ Exemple de composition de stratégies
  - ▶ Notion d'univers
- ▶ Conclusion

# Contributions théoriques : Partie 1

**Hierarchie de treillis** : formalisation complète et unifiée de la programmation par contraintes.

## Chapitre 2 : Historique des langages de programmation par contraintes.

- ▶ Programmation logique (70').
- ▶ Programmation logique concurrente et par contraintes (80' et 90').
- ▶ Langages de modélisation (90' et 00').
- ▶ Travaux plus récents sur les stratégies d'exploration (00' et 10') :
  - ▶ Search combinators (Schrijvers and al. 11')
  - ▶ Tor predicates (Schrijvers and al. 14')
  - ▶ ClpZinc (Martinez and al. 15')

## Contributions théoriques : Partie 2

- ▶ *Spacetime programming* : Instanciation de la hiérarchie de treillis.
- ▶ **Étude de la causalité et sémantique** d'un langage synchrone sur les treillis.
- ▶ **Cas d'études** : composition assistée par ordinateur et *model checking*.

	Esterel	Oz	<i>Spacetime</i>
Hiérarchie temporelle	✓	X	✓
Hiérarchie spatiale	$L_0$	$L_0, L_1, \dots, L_n$	$L_0, L_1, \dots, L_n$
<i>Backtracking</i>	X	✓	✓

## Contributions pratiques

- ▶ **L'implémentation** : [github.com/ptal/bonsai](https://github.com/ptal/bonsai)
- ▶ Étend la bibliothèque synchrone SugarCubes (Susini, 01').
- ▶ **Abstraction en treillis** du solveur Choco ( $L_3$ ).
- ▶ **Les expérimentations** montrent un temps d'exécution raisonnable comparé au module d'exploration de Choco.

	Temps (secondes)		
	Choco	Spacetime	$\frac{Spacetime}{Choco}$
	Première solution		
Latin square (40)	2.94s	3.27s	1.11
Latin square (50)	7.95s	8.78s	1.10
	Toutes les solutions		
N-Queens (13)	6.00s	16.04s	2.67
N-Queens (14)	32.13s	91.00s	2.83

# Perspectives

## Hiérarchie de treillis

- ▶ **Établir les propriétés** de la hiérarchie de treillis comme la terminaison, exhaustivité, compacité.
- ▶ **Étendre la hiérarchie** : recherche locale, domaines infinis.

## *Spacetime programming*

- ▶ **Implémentation** de la notion d'univers et de causalité dans le compilateur de *spacetime*.
- ▶ **Extensions de spacetime** : processus d'ordre supérieur, structure de données.
- ▶ **Combiner spacetime** : *model checking*, systèmes de réécriture.

Merci pour votre attention.



 [github.com/ptal/bonsai](https://github.com/ptal/bonsai)