Spacetime programming

A Synchronous Language for Composable Search Strategies

Pierre Talbot

(pierre.talbot@univ-nantes.fr)

University of Nantes
LS2N Laboratory

8th October 2019

UNIVERSITÉ DE NANTES

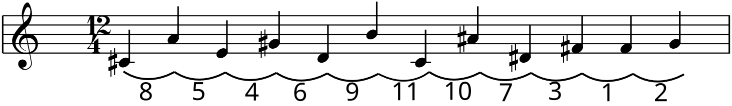
Constraint programming

“Holy grail of computing”

- Declarative paradigm for solving combinatorial problems.
- We state the problem and let the computer solve it for us.


An example of constraint problem

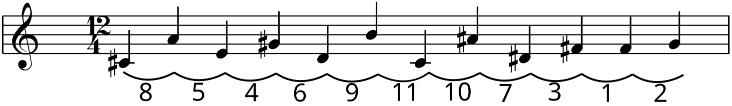
Find a series of 12 notes such that every note and every interval between two successive notes are distinct.



- We only state **what** constraints the solution should verify.
- We do not say **how** to find the solution.

Model of the "All-Interval Series" problem

Find a series of 12 notes such that every note and every interval between two successive notes are distinct.



Model in MiniZinc:

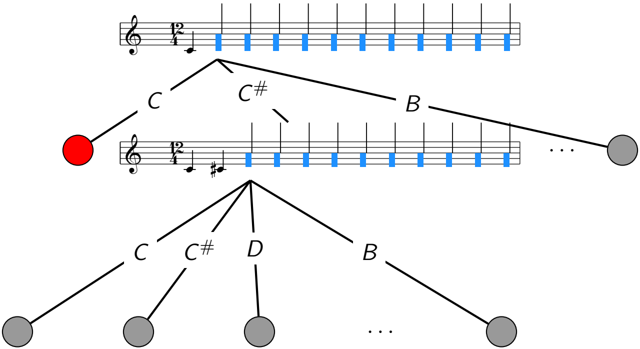
```
int: n = 12;
array[1..n] of var 1..n: pitches;
array[1..n−1] of var 1..n − 1: intervals;
constraint forall(i in 1..n − 1)
                  ( intervals [i] = abs(pitches[i+1] − pitches[i]));
constraint alldifferent (pitches);
constraint alldifferent ( intervals );

solve satisfy;
```

How to find a solution?

NP-complete nature

- Try every combination until we find a solution.
- Backtracking algorithm builds and explores a search tree.



Problem

Holy grail?

- Search tree is often too huge to find a solution in a reasonable time.
- **Search strategies are crucial** to improve efficiency.
- Search strategies are often problem-dependent so we need to try and test (empirical evaluation).

State of the art

1. **Languages** (Prolog, MiniZinc,...): Clear and compact description but limited amount of pre-defined strategies or compositionality issues.
2. **Libraries** (Choco, GeCode,...): Highly customizable and efficient but complex software, hard to understand and time-consuming.

- Composing strategies is impossible or hard in both cases.

Lack of abstraction for expressing, composing and extending search strategies.

Proposal

A language named spacetime programming

Inspired by synchronous programming (Esterel) and timed concurrent constraint programming (TCC).

Key idea: Logical time to combine concurrency and backtracking.

- **Strategy = Process** exploring a state space. We compose strategies as we compose processes.
- **Logical time** allows us to coordinate the strategies exploring the search tree.

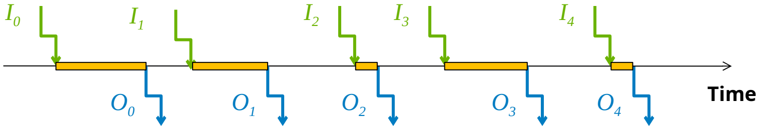
Outline

- Introduction
- Synchronous programming
- Spacetime programming
  - Syntax and model of computation
  - Composition of search strategies
- Conclusion

Synchronous paradigm

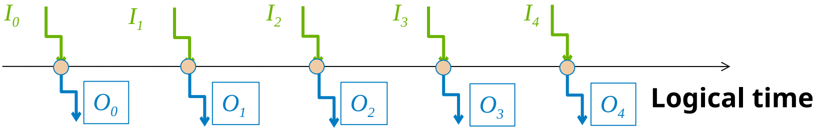
- Invented in the 80' to deal with reactive system subject to many (simultaneous) inputs.
- Continuous interaction with the environment.

Dividing the execution into logical instants:



Synchronous paradigm

Synchronous hypothesis: An instant does not take time:



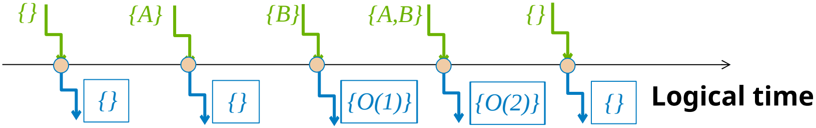
- Strong guarantee of **determinism**: for one set of inputs, only one output possible (causality analysis).

An example in Esterel (*Berry et al.*, 92')

Emit O as soon as A and B arrived, and count the occurrences of 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
```

(Note that `await` contains a `pause` statement).



Outline

- Introduction
- Synchronous programming
- Spacetime programming
  - Syntax and model of computation
  - Composition of search strategies
- Conclusion

Main features of spacetime

- Replace Boolean variables of Esterel with **arbitrary lattice variables**.
  $\Rightarrow$ A constraint problem can be represented as a lattice.
- **Model of computation**:
  - The state space is stored in a queue of nodes.
  - A node of the search tree is explored in exactly one logical instant.
- Behavioral semantics of spacetime with guarantees that spacetime programs are **reactive, deterministic and extensive** functions.

Model of computation through an example

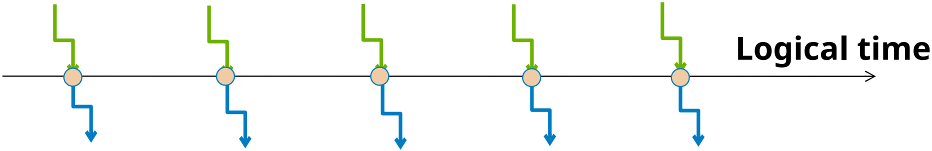
Counting the number of right branches (called "discrepancies") in a tree of depth 2 at maximum.

```
single_space LMax n = new LMax(0);   } Counter of nodes.
world_line LMax d = new LMax(0);     } Counter of discrepancies.
loop
  n ← n + 1;
  when depth < 2 then
    space nothing end;               } Creating two children nodes.
    space d ← d + 1 end;
  end
  pause
end
```

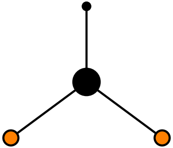
LMax is the lattice of increasing integers.

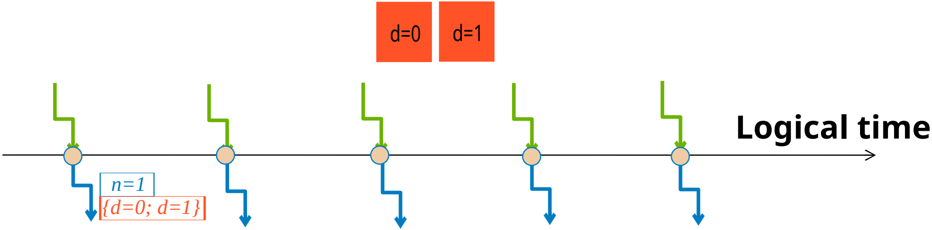
```
single_space LMax n = new LMax(0);
world_line LMax d = new LMax(0);
loop
  n ← n + 1;
  when depth < 2 then
    space nothing end;
    space d ← d + 1 end;
  end
  pause
end
```

Stack with an empty node:

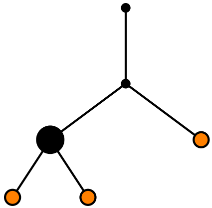


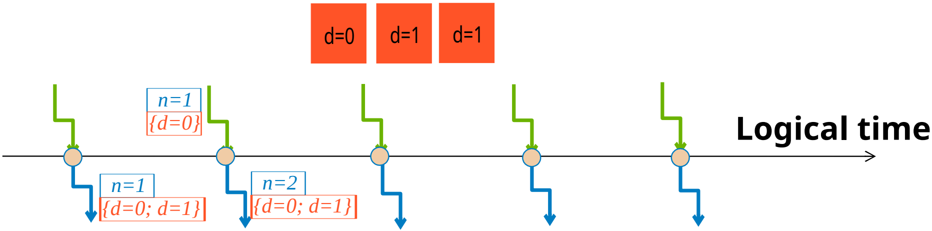
```
single_space LMax n = new LMax(0);
world_line LMax d = new LMax(0);
loop
  n ← n + 1;
  when depth < 2 then
    space nothing end;
    space d ← d + 1 end;
  end
  pause
end
```



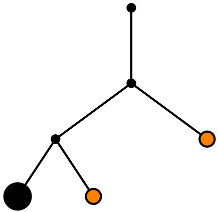


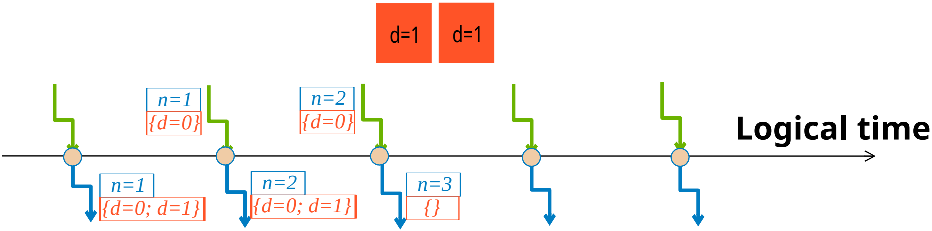
```
single_space LMax n = new LMax(0);
world_line LMax d = new LMax(0);
loop
  n ← n + 1;
  when depth < 2 then
    space nothing end;
    space d ← d + 1 end;
  end
  pause
end
```



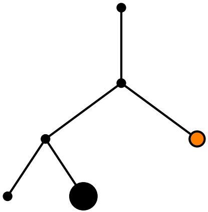


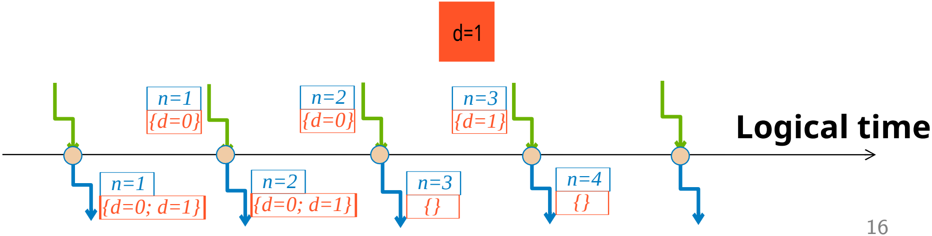
```
single_space LMax n = new LMax(0);
world_line LMax d = new LMax(0);
loop
  n ← n + 1;
  when depth < 2 then
    space nothing end;
    space d ← d + 1 end;
  end
  pause
end
```



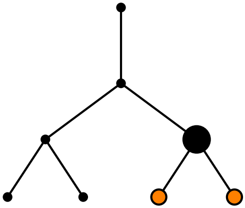


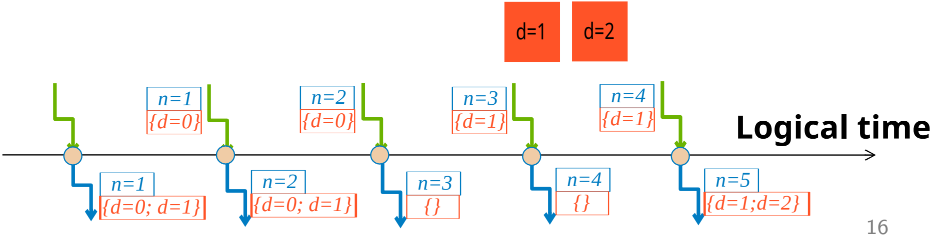
```
single_space LMax n = new LMax(0);
world_line LMax d = new LMax(0);
loop
  n ← n + 1;
  when depth < 2 then
    space nothing end;
    space d ← d + 1 end;
  end
  pause
end
```





```
single_space LMax n = new LMax(0);
world_line LMax d = new LMax(0);
loop
  n ← n + 1;
  when depth < 2 then
    space nothing end;
    space d ← d + 1 end;
  end
  pause
end
```

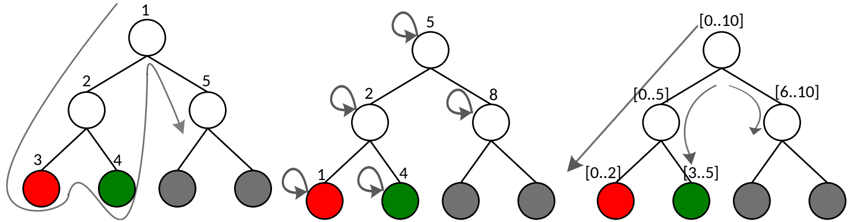




Spacetime attributes

We define three spacetime attributes to locate a variable in time and space:

- `single_space`: variable **global** to the search tree.
- `single_time`: variable **local** to an instant/node.
- `world_line`: **backtrackable** variable / local to a path in the search tree.



Syntax of spacetime

$\langle p, q, \ldots \rangle ::=$		**Communication fragment**
$\mid$	*spacetime Type* $x = e$	(variable declaration)
$\mid$	**when** x **|=** y **then** p **else** q **end**	(ask)
$\mid$	$f(args)$	(host function call)
$\mid$		**Synchronous fragment**
$\mid$	**par** p **||** q **end**	(disjunctive parallel)
$\mid$	**par** p **<>** q **end**	(conjunctive parallel)
$\mid$	p **;** q	(sequence)
$\mid$	**loop** p **end**	(loop)
$\mid$	**pause**	(delay)
$\mid$		**Search fragment**
$\mid$	**space** p **end**	(create a branch)
$\mid$	**prune**	(prune a branch)

Outline

Composition of search strategies

Each process generates a sequence of branches that can be combined in various ways:

- A process without `prune` or `space` generates an empty sequence (identity element).
- `prune` generates a single pruned branch.
- `space` p generates a single branch.
- p `;` q: concatenation of the branches of p and q.
- p `||` q: pairwise union of the branches.
- p `<>` q: pairwise intersection of the branches.

$$(\texttt{prune}\ ;\ \texttt{space}\ p)\ \texttt{<>}\ (\texttt{space}\ q\ ;\ \texttt{space}\ r) \rightarrow \langle \texttt{prune}, \texttt{space}\ (p\texttt{<>}r)\rangle$$

A recipe to program your search strategy

We create different sub-strategies that we assemble next:

- Create the “raw state space” of a CSP.
- Propagate the nodes in this CSP.
- Bound the depth.
- Assemble!

Sequential composition

Creating the state space of a constraint satisfaction problem:

```
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
```

Class fields with *spacetime* attributes

Java constructor

Branching strategy

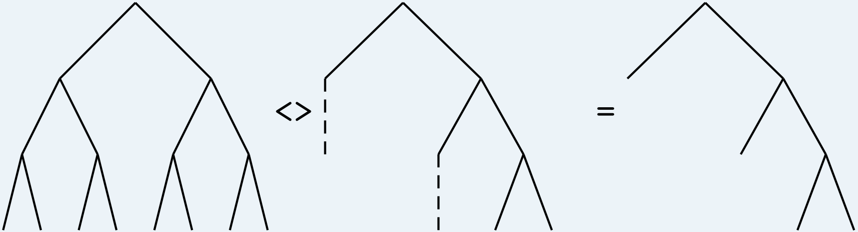
$x \leq v \vee x > v$

Propagation

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

$(ES = \mathit{false} \models \mathit{true} \models \mathit{unknown})$

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

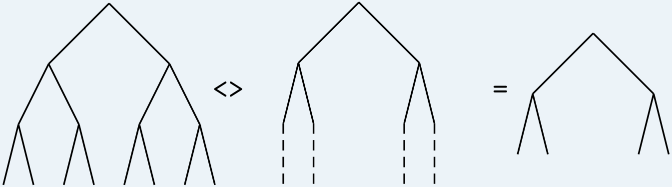


Depth-bounded search

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

Prune the branches when the limit is reached.

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



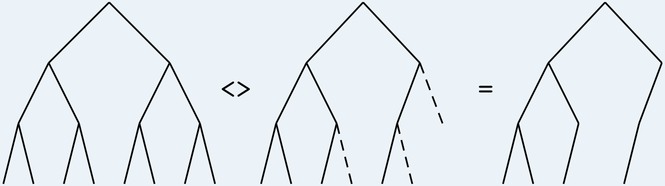
Discrepancy-bound search

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

Left branch

Right branch

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



Combining trees by intersection

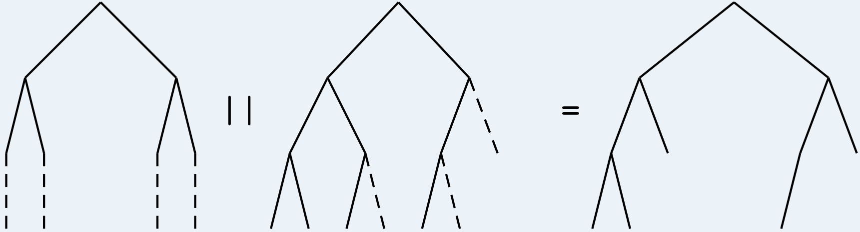
We can compose depth-bounded and discrepancy-bounded search by intersection:

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

Combining trees by union

We can compose depth-bounded and discrepancy-bounded search by union:

```
par  bound_depth(2) || bound_discrepancy(1)  end
```



Summary

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

- **Communication** among strategies through the variables `domains` and `constraints`.
- **Compositional and reusable**: each strategy is specified independently.

Outline

Implementation and experiments

- Compiler implemented in Rust and open-source: `github.com/ptal/bonsai`.
- The runtime (in Java) is inspired by SugarCubes (Susini, 01') and ReactiveML (Mandel et al., 06').
- **Lattice abstraction** of the constraint solver Choco.

Problem	Spacetime	Choco	Factor
14-Queens	89.9s (62020n/s)	30.6s (182218n/s)	2.9
15-Queens	528.2s (60972n/s)	185.2s (173816n/s)	2.85
Golomb Ruler 11	40.1s (14186n/s)	27.2s (20888n/s)	1.47
Golomb Ruler 12	425.8s (10871n/s)	279.8s (16541n/s)	1.52
Latin Square 75	61.2s (73n/s)	57.9s (77n/s)	1.06
Latin Square 90	150.3s (44n/s)	147.8s (45n/s)	1.02

(n/s = nodes per second)

Conclusion

- Spacetime is a language to program and combine search strategies, combining concurrency and backtracking, inspired by:
  - (Timed) concurrent constraint programming (Saraswat et al., 89')
  - Synchronous programming, Esterel (Berry et al., 92')
- Spacetime programs are **reactive, deterministic and extensive**.

Conclusion

- Spacetime is a language to program and combine search strategies, combining concurrency and backtracking, inspired by:
  - (Timed) concurrent constraint programming (Saraswat et al., 89')
  - Synchronous programming, Esterel (Berry et al., 92')
- Spacetime programs are **reactive, deterministic and extensive**.

What's next

- Merge **deep guards** of logic programming with **time hierarchy** of synchronous programming.
  $\Rightarrow$ To program restart-based search strategies / nested search.
- Go beyond the scope of constraint programming.

Thanks! `github.com/ptal/bonsai`