Abstract Constraint Programming on GPU

Talk at National University of Singapore, Programming Language Innovation Lab (Prof. Adams group)

Pierre Talbot

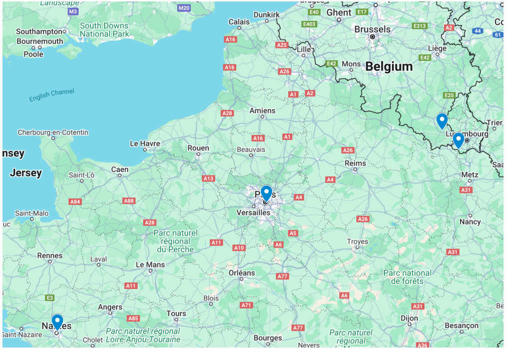
pierre.talbot@uni.lu

https://ptal.github.io

21st January 2026

University of Luxembourg

Who am I?



- 2014: Master in CS/PL, Sorbonne University, Paris
- 2014–2018: Ph.D., Sorbonne University, Paris
  - *Spacetime Programming: A Synchronous Language for Constraint Search.*
- 2018–2019: Postdoc, University of Nantes.
  - *Abstract Domains for Constraint Programming.*
- 2020–2023: Postdoc, University of Luxembourg
  - *A Lattice-Based Approach for GPU Programming.*
- 2023–: Research scientist, University of Luxembourg.
  - *Abstract Satisfaction and Parallel Computing.*

Why Am I Here?

Fixed-Point-Oriented Programming: A Concise and Elegant Paradigm

PROGRAMMING LANGUAGE INNOVATION LAB, National University of Singapore, Singapore

Fixed-Point-Oriented Programming (FPOP) is an emerging paradigm designed to streamline the implementation of problems involving self-referential computations. These include graph algorithms, static analysis, parsing, and distributed computing—domains that traditionally require complex and tricky-to-implement work-queue algorithms. Existing programming paradigms lack direct support for these inherently fixed-point computations, leading to inefficient and error-prone implementations.

This white paper explores the potential of the FPOP paradigm, which offers a high-level abstraction that enables concise and expressive problem formulations. By leveraging structured inference rules and user-directed optimizations, FPOP allows developers to write declarative specifications while the compiler ensures efficient execution. It not only reduces implementation complexity for programmers but also enhances adaptability, making it easier for programmers to explore alternative solutions and optimizations without modifying the core logic of their program.

We demonstrate how FPOP simplifies algorithm implementation, improves maintainability, and enables rapid prototyping by allowing problems to be clearly and concisely expressed. For example, the graph distance problem can be expressed in only two executable lines of code with FPOP, while it takes an order of magnitude more code in other paradigms. By bridging the gap between theoretical fixed-point formulations and practical implementations, we aim to foster further research and adoption of this paradigm.

[...] array-based problems are naturally solved by iteration and tree-based problems are naturally solved by recursion, what is the natural paradigm for graph-based problems?

Why Am I Here?

Fixed-Point-Oriented Programming: A Concise and Elegant Paradigm

PROGRAMMING LANGUAGE INNOVATION LAB, National University of Singapore, Singapore

Fixed-Point-Oriented Programming (FPOP) is an emerging paradigm designed to streamline the implementation of problems involving self-referential computations. These include graph algorithms, static analysis, parsing, and distributed computing—domains that traditionally require complex and tricky-to-implement work-queue algorithms. Existing programming paradigms lack direct support for these inherently fixed-point computations, leading to inefficient and error-prone implementations.

This white paper explores the potential of the FPOP paradigm, which offers a high-level abstraction that enables concise and expressive problem formulations. By leveraging structured inference rules and user-directed optimizations, FPOP allows developers to write declarative specifications while the compiler ensures efficient execution. It not only reduces implementation complexity for programmers but also enhances adaptability, making it easier for programmers to explore alternative solutions and optimizations without modifying the core logic of their program.

We demonstrate how FPOP simplifies algorithm implementation, improves maintainability, and enables rapid prototyping by allowing problems to be clearly and concisely expressed. For example, the graph distance problem can be expressed in only two executable lines of code with FPOP, while it takes an order of magnitude more code in other paradigms. By bridging the gap between theoretical fixed-point formulations and practical implementations, we aim to foster further research and adoption of this paradigm.

> *Examples of such problems include parsing, static analysis, type-checking, graph algorithms, automata minimization, and distributed computing [...]*

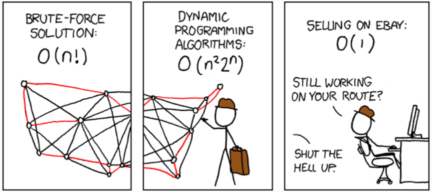
To this list, I'd like to add today *constraint reasoning* and *parallel programming*.

My Research in a Nutshell!

I research on the “fusion” of...

Constraint reasoning + Abstract interpretation

(and lattice theory)



that gives us abstract satisfaction.

My Research in a Nutshell!

I research on the “fusion” of...

WHY?

Accelerate constraint solving

HOW?

- Combining constraint solvers
- Constructing sound solving procedure over complex domains
- Constraint solving on GPUs

that gives us abstract satisfaction.

My Research in a Nutshell!

I research on the "fusion" of...

WHY?

Accelerate constraint solving

HOW?

- Combining constraint solvers
- Constructing sound solving procedure over complex domains
- Constraint solving on GPUs $\Leftarrow$ TODAY

that gives us abstract satisfaction.

TEAM

I have the pleasure to co-supervise and collaborate with several Master students, Ph.D. candidates and postdocs.


Pierre Talbot


Hedieh Haddad


Manuel Combarro


Yi-Nung Tsao


Tobias Fischbach


Hakan Hasan

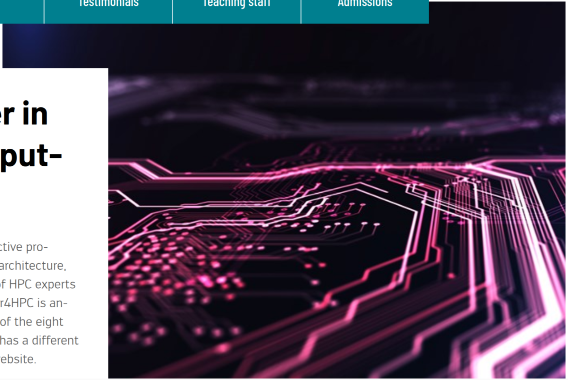

Wei Huang


Anisa Meta

- Hasan Hakan, Ph.D. candidate, *TBD*, 2025-2028.
- Yi-Nung Tsao, Ph.D. candidate, *Verification of Neural Networks by Abstract Interpretation*, 2023-2027.
- Manuel Combarro, Ph.D. candidate, *Multiobjective Constraint Programming*, 2023-2026.
- Hedieh Haddad, Ph.D. candidate, *Hyperparameter Optimization of Constraint Solver*, 2023-2026.
- Tobias Fischbach, Ph.D. candidate, *Optimization of Quantum Circuits*, 2023-2026.
- Wei Huang, Master student, *Improving Fixpoint Loop in Turbo* (master thesis), March–August 2026.
- Anisa Meta, Master student, *GPU-based Inprocessing in Turbo* (master thesis), February–July 2026.

Study Programmes

Master in High Performance Computing



Your outstanding career in high-performance computing

The Master in High Performance Computing (MHPC) is a distinctive programme at the intersection of parallel programming, hardware architecture, and artificial intelligence. We are training the next generation of HPC experts in Luxembourg and Europe. Besides to the MHPC, the EUMaster4HPC is another programme where students earn a dual degree from two of the eight universities of the EUMaster4HPC consortium. EUMaster4HPC has a different application procedure, so be sure to check out the dedicated website.

Lattice Theory for Parallel Programming

Description: Lattice theory is one of the most useful mathematical theories to describe and prove
of computer science starting with denotational semantics (what is a program from the mathematic
recently in parallel and distributed computing with conflict-free replicated data types (CRDTs) and
incomplete view of the global state). CRDTs are widely used to program highly-available services f

This course is given in the *Master in High Performance Computing* at the University of Luxembourg w
assistant).

The course is self-contained, only basic knowledge of set theory and logic is necessary. Half of the
lattice theory (given by myself).

- Lectures on lattice theory (by Bruno) are available here
- Lecture 1: Overview of the Course [pdf]
- Lecture 2: Conflict-free Replicated Data Type [pdf] with a laboratory [pdf]
- Lecture 3: More Conflict-free Replicated Data Type [pdf] with a laboratory [pdf]
- Lecture 4: Parallel Lattice Programming [pdf]
- Lecture 5: Abstract Satisfaction [pdf]
- Lecture 6: Neural Network Verification [pdf]
- Lecture 7: Abstract Interpretation [pdf]

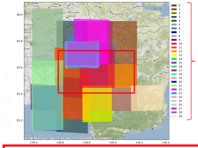
Constraint Programming

Constraint Programming

Constraint programming: FOL without quantifiers, $\mathbb{U} = \mathbb{Z}$ and arithmetic constraints.

- **Declarative paradigm**: specify your problem and let the computer solves it for you.
- **Many applications**: scheduling, bin-packing, hardware design, satellite imaging, ...
- **Constraint programming** is one approach to solve such combinatorial problems.
- Other approaches include SAT, linear programming, SMT, MILP, ASP,...

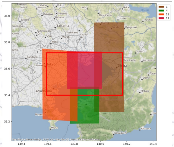
Satellite image mosaic



After a query with certain parameters:

N = 30 satellite images

Find the cover with the minimum number of images (NP-Hard)

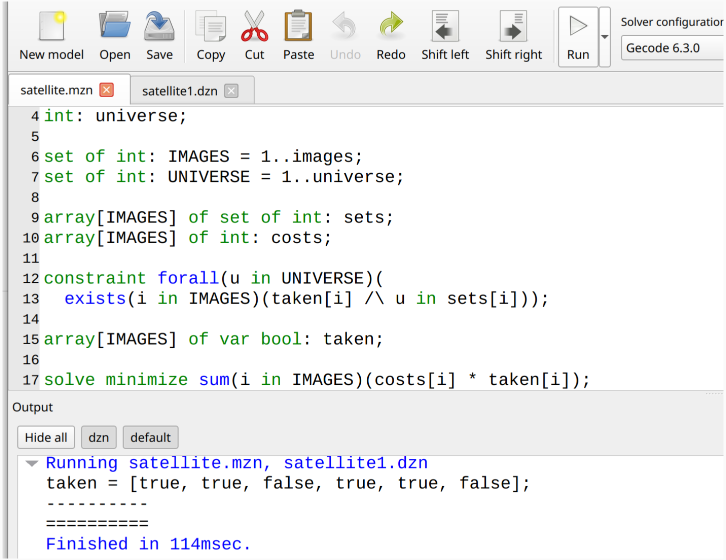


Build the mosaic

[1]

[1]Combarro et al., Constraint Model for the Satellite Image Mosaic Selection Problem, CP 2023

Constraint model of satellite imaging in MiniZinc:



Constraint Network

Constraint Network

Let X be a finite set of variables and C be a finite set of constraints.
A *constraint network* is a pair $P = \langle d, C \rangle$ such that $d \in X \to Itv$ is the *domain* of the variables where Itv is the set of intervals.

Note: It is just a "format" to represent quantifier-free logical formulas where variables have bounded domains.

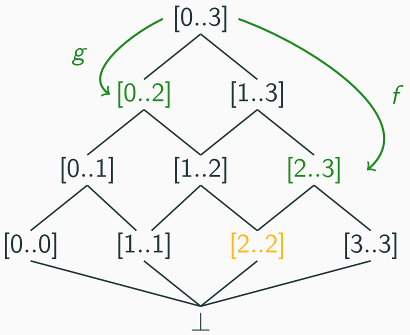
Example

$$\langle \{x \mapsto [0,2], y \mapsto [2,3]\}, \{x \leq y - 1\} \rangle$$

A solution is $\{x \mapsto 0, y \mapsto 2\}$.

Parallel Model of Computation

Parallel Model of Computation



- $f(x) \triangleq x \sqcap [2..\infty]$ models the constraint $x \geq 2$.
- $g(x) \triangleq x \sqcap [-\infty..2]$ models the constraint $x \leq 2$.
- Parallel execution: `f || g = [2..2]`

Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

Memory:

$$\boxed{x = [-\infty, \infty]}$$

Propagators:

$$
\begin{array}{lll}
 & x \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\
\parallel & x \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!])
\end{array}
$$

Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

Memory:

$$\boxed{x = [-\infty, ?]}$$

Propagators:

$$\begin{array}{lll} & x \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & x \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

Issue 1: data race? Parallel update of the same variable: upper bound of x.

Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

Memory:

$\boxed{x = [-\infty, 5]}$

Propagators:

$$\begin{array}{llr} & \boxed{x} \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & \boxed{x} \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

Issue 1: data race? Parallel update of the same variable: upper bound of x.

$\Rightarrow$ **Solution**: Use atomic load and store!

Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4]\!] \mid\mid \mathcal{I}[\![x \leq 5]\!]$

Memory:

$$\boxed{x = [-\infty, 5]}$$

Propagators:

$$\begin{array}{lll} & \boxed{x} \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\ \mid\mid & \boxed{x} \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

Issue 1: data race? Parallel update of the same variable: upper bound of x.

$\Rightarrow$ **Solution**: Use atomic load and store!

$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

Memory:

$$\boxed{x = [-\infty, 5]}$$

Propagators:

$$\begin{array}{lll} & x \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & x \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

Issue 1: data race? Parallel update of the same variable: upper bound of x.

$\Rightarrow$ **Solution**: Use atomic load and store!

$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

Issue 2: nondeterminism? x can be equal to $[-\infty, 4]$ or $[-\infty, 5]$ depending on the order of execution.

Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4]\!] \mid\mid \mathcal{I}[\![x \leq 5]\!]$

Memory:

$x = [-\infty, 4]$

Propagators:

$$
\begin{array}{llll}
 & x \leftarrow [-\infty, 4] & & (\mathcal{I}[\![x \leq 4]\!]) \\
\mid\mid & x \leftarrow [-\infty, 5] & & (\mathcal{I}[\![x \leq 5]\!])
\end{array}
$$

Issue 1: data race? Parallel update of the same variable: upper bound of x.
$\Rightarrow$ **Solution**: Use atomic load and store!
$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

Issue 2: nondeterminism? x can be equal to $[-\infty, 4]$ or $[-\infty, 5]$ depending on the order of execution.
$\Rightarrow$ **Solution**: Use a fixpoint loop.

Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

Memory:

$$\boxed{x = [-\infty, 5]}$$

Propagators:

$$\begin{array}{lll} & x \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & x \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

Issue 1: data race? Parallel update of the same variable: upper bound of x.
$\Rightarrow$ **Solution**: Use atomic load and store!
$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

Issue 2: nondeterminism? x can be equal to $[-\infty, 4]$ or $[-\infty, 5]$ depending on the order of execution.
$\Rightarrow$ **Solution**: Use a fixpoint loop.

Issue 3: progress? What if $\mathcal{I}[\![x \leq 5]\!]$ is always "winning"?

Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

Memory:

$$x = [-\infty, \boxed{4}]$$

Propagators:

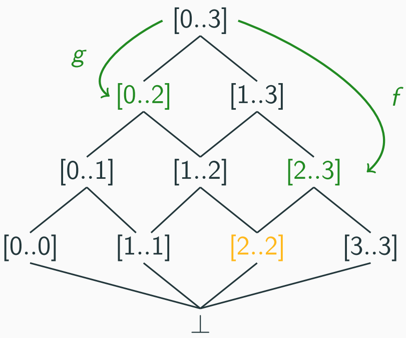
$$\begin{array}{lll} & \boxed{x} \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & \boxed{x} \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

Issue 1: data race? Parallel update of the same variable: upper bound of x.
$\Rightarrow$ **Solution**: Use atomic load and store!
$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

Issue 2: nondeterminism? x can be equal to $[-\infty, 4]$ or $[-\infty, 5]$ depending on the order of execution.
$\Rightarrow$ **Solution**: Use a fixpoint loop.

Issue 3: progress? What if $\mathcal{I}[\![x \leq 5]\!]$ is always "winning"?
$\Rightarrow$ **Solution**: Write in the memory only if the value is strictly lower ($\lceil x \rceil = v$ iff $v < \lceil x \rceil$).

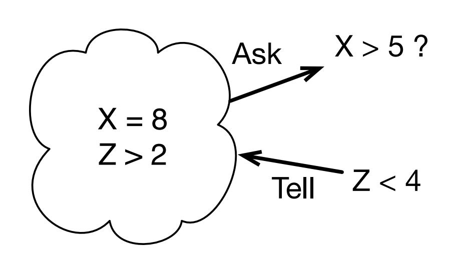
Parallel Model of Computation



But what should be the properties of f and g? How to design such functions?

Concurrent Constraint Programming

Concurrent constraint programming (CCP) is a *process calculus* introduced in the eighties[2].
Two main operations: ask and tell.



Conceptual idea: allow to compute with partial information; replace the "Von Neumann" memory model by a constraint store.

[2] V. A. Saraswat and M. Rinard, *Concurrent constraint programming* (POPL-89)

Syntax of CCP

Let $x, x_1, \ldots, x_n \in X$ be variables, $c, c_1, \ldots$ be constraints, p a predicate name.

$$
\begin{array}{rcll}
\langle P, Q \rangle & ::= & \sum_{i \in I} ask(c_i) \; ? \; P_i & \textit{sum statement} \\
& | & tell(c) & \textit{tell statement} \\
& | & \exists x, \; P & \textit{local statement} \\
& | & P \parallel Q & \textit{parallel composition} \\
& | & p(x_1, \ldots, x_n) & \textit{predicate call}
\end{array}
$$

$$
\begin{array}{rcll}
\langle A, B \rangle & ::= & p(x_1, \ldots, x_n) = P & \textit{predicate definition} \\
& | & A \; B & \textit{list of predicates}
\end{array}
$$

Example

$$
\begin{array}{l}
\exists x, y, z, \\
\quad ask(y = 1) \; ? \; tell(z > 10) \\
\parallel ((ask(x = 0) \; ? \; tell(y = 1)) \; + \; (ask(x = 1) \; ? \; tell(y = 2)))
\end{array}
$$

Sketch of Semantics

Definitions

- A *configuration* is a pair $\langle P, C\rangle$ where P is a CCP process to execute, and C is a store of constraint.
- A "step of execution" is given by a relation $\langle P, C\rangle \to \langle P', C'\rangle$.

$$\textsc{Tell}\quad \langle \mathit{tell}(c), C\rangle \to \langle \mathit{tell}(c), C \cup \{c\}\rangle$$

$$\textsc{Par-Left}\quad \frac{\langle P, C\rangle \to \langle P', C'\rangle}{\langle P \parallel Q, C\rangle \to \langle P' \parallel Q, C'\rangle}$$

$$\textsc{Par-Right}\quad \frac{\langle Q, C\rangle \to \langle Q', C'\rangle}{\langle P \parallel Q, C\rangle \to \langle P \parallel Q', C'\rangle}$$

Main Properties

- *Monotonicity*: $\rightarrow$ is monotone over the store of constraints, in particular it means:
  - If $ask(c)$ is true in a store C then it is true in every store C' such that $C \subseteq C'$.
- *Extensive*: $\rightarrow$ is extensive over the store of constraints (we cannot remove information).
- *Closure operator*: $\rightarrow^*$ is a closure operator over the store.[3]
- *Restartable*: Suppose we perform a partial execution $\langle P, C \rangle \rightarrow \ldots \rightarrow \langle P', C' \rangle$, then we can restart the execution from $\langle P, C' \rangle$ (and obtain the same result).

[3]Supposing the branches of the sum are all disjoints—called *determinate CCP*.

Parallel CCP

Parallel Concurrent Constraint Programming (PCCP)

Observation: CCP lacks a proper connection to parallel architecture, and had limited impact despite a beautiful theory.
We worked on that by simplifying the language (no recursion) and using lattice to define constraint system[4].

[4] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU* (AAAI 2022)

Parallel Concurrent Constraint Programming (PCCP)

Observation: CCP lacks a proper connection to parallel architecture, and had limited impact despite a beautiful theory.
We worked on that by simplifying the language (no recursion) and using lattice to define constraint system[4].

Syntax of PCCP

Let $x, y, y_1, \dots, y_n \in X$ be variables, L a lattice, f a monotone function, and b a Boolean variable of type $\langle \{true, false\}, \Leftarrow \rangle$:

$\langle P, Q \rangle ::=$	$\texttt{if } b \texttt{ then } P$	*ask statement*
$\mid$	$x \leftarrow f(y_1, ..., y_n)$	*tell statement*
$\mid$	$\exists x{:}\mathrm{L},\ P$	*local statement*
$\mid$	$P \parallel Q$	*parallel composition*

[4] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU* (AAAI 2022)

Examples: Minimum and Constraint

Let $\mathtt{ZUB}$ the lattice of increasing integers, and $\mathtt{ZLB}$ the lattice of decreasing integers.

Minimum

$\exists m : \mathtt{ZUB},\ m \leftarrow x_1 \parallel \ldots \parallel m \leftarrow x_n$ (unfolded for-loop)

$x + y \leq c$ constraint

Suppose the variables x and y are defined by four variables xl, xu, yl, yu modelling the intervals $[xl, xu]$ and $[yl, yu]$.

$$[\![x + y \leq c]\!] \triangleq xu \leftarrow c - yl \parallel yu \leftarrow c - xl$$

(see lecture on "abstract satisfaction").

Denotational Semantics

- A PCCP process is a reductive and monotone function over a Cartesian product $Store = L_1 \times \ldots \times L_n$ storing the values of all local variables.
- Since we do not have recursion, we know at compile-time the number of variables.
- Let $Proc$ be the set of all PCCP processes.

Denotational Semantics

- A PCCP process is a reductive and monotone function over a Cartesian product $Store = L_1 \times \ldots \times L_n$ storing the values of all local variables.
- Since we do not have recursion, we know at compile-time the number of variables.
- Let *Proc* be the set of all PCCP processes.

Denotational Semantics

We define a function $\mathcal{D} : Proc \to (Store \to Store)$:

$$
\begin{aligned}
&\mathcal{D}(x \leftarrow f(y_1, .., y_n)) \triangleq \lambda s.s[x \mapsto s(x) \sqcap f(s(y_1), .., s(y_n))] \\
&\mathcal{D}(\texttt{if } b \texttt{ then } P) \triangleq \lambda s.(\!|\ s(b) \mathrel{?} \mathcal{D}(P)(s) \mathrel{\circ\!\!\circ} s\ |\!) \\
&\mathcal{D}(P \parallel Q) \triangleq \mathcal{D}(P) \sqcap \mathcal{D}(Q)
\end{aligned}
$$

Executing the program: **gfp** $\mathcal{D}(P)$.

Sequential Computation = Parallel Computation

We obtain the same result if we execute P in parallel or if we replace all parallel $\|$ by a sequential operator ; (a transformation we write $\mathtt{seq}\ P$) defined as follows:

$$\mathcal{D}(P\ ;\ Q) \triangleq \mathcal{D}(Q) \circ \mathcal{D}(P)$$

Let **fix** f be the set of fixpoints of a function f.

Theorem (Equivalence Between Sequential and Parallel Operators)

$\mathbf{fix}\ \mathcal{D}(\mathtt{seq}\ P) = \mathbf{fix}\ \mathcal{D}(P)$

Conclusion

C++ Abstraction: Lattice Land Project

`lattice-land` is a collection of libraries abstracting our parallel model.

It provides various data types and fixpoint engine:

- `ZLB`, `ZUB`: increasing/decreasing integers.
- `B`: Boolean lattices.
- `VStore`: Array (of lattice elements).
- `IPC`: Arithmetic constraints.
- `GaussSeidelIteration`: Sequential CPU fixed point loop.
- `AsynchronousIteration`: GPU-accelerated fixed point loop.
- ...

```
void max(int tid, const int* data, ZLB& m) {
  m.tell(data[tid]);
}
AsynchronousIteration::fixpoint(max);
```

https://github.com/lattice-land

Conclusion: Theoretical Parallel Model of Computation

Data races occur rarely, so we should avoid working so much to avoid them.

Properties of the model

A Variant of Concurrent Constraint Programming on GPU (AAAI 2022)[5].

- Correct: Proofs that $P; Q \equiv P || Q$, parallel and sequential versions produce the same results.
- Restartable: Stop the program at any time, and restart on partial data.
- Weak memory consistency: Very few requirements on the underlying memory model $\Rightarrow$ wide compatibility across hardware, unlock optimization.

[5] `https://ptal.github.io/papers/aaai2022.pdf`

Conclusion: Practical Implementation

A GPU-based Constraint Programming Solver (AAAI 2026)[6].

- Simple: solving algorithms from 50 years ago.
  $\Rightarrow$ no global constraints, nogoods learning, lazy clause generation, restart strategies, event-based propagation, trailing or recomputation-based state restoration and domain consistency.
- Efficient: Almost on-par with Choco (algorithmic optimization VS hardware optimization).
- Many possible optimizations to improve the efficiency, but need to be redesigned for GPU.

`https://github.com/ptal/turbo`

[6] `https://ptal.github.io/papers/aaai2026.pdf`