Hero Net Interpreter

Final presentation

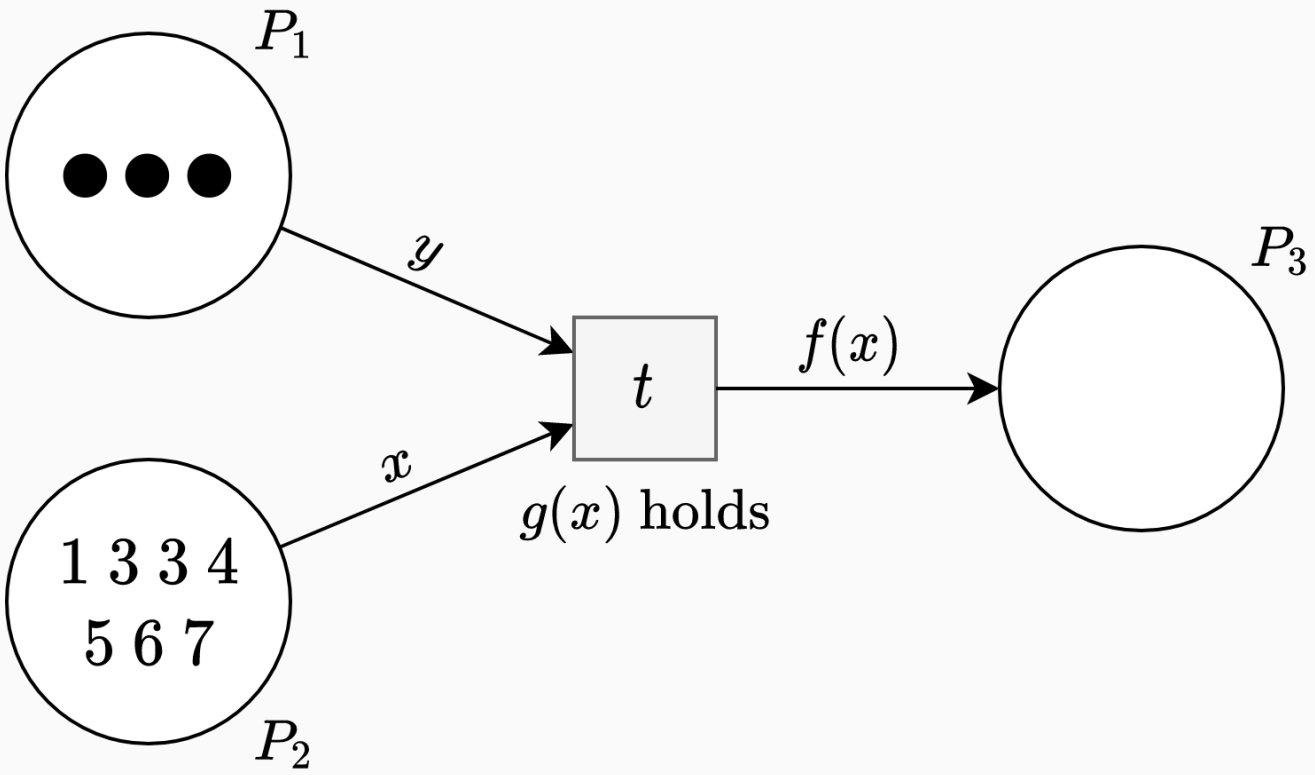
Martin Jérémie

September 16, 2020

University of Geneva

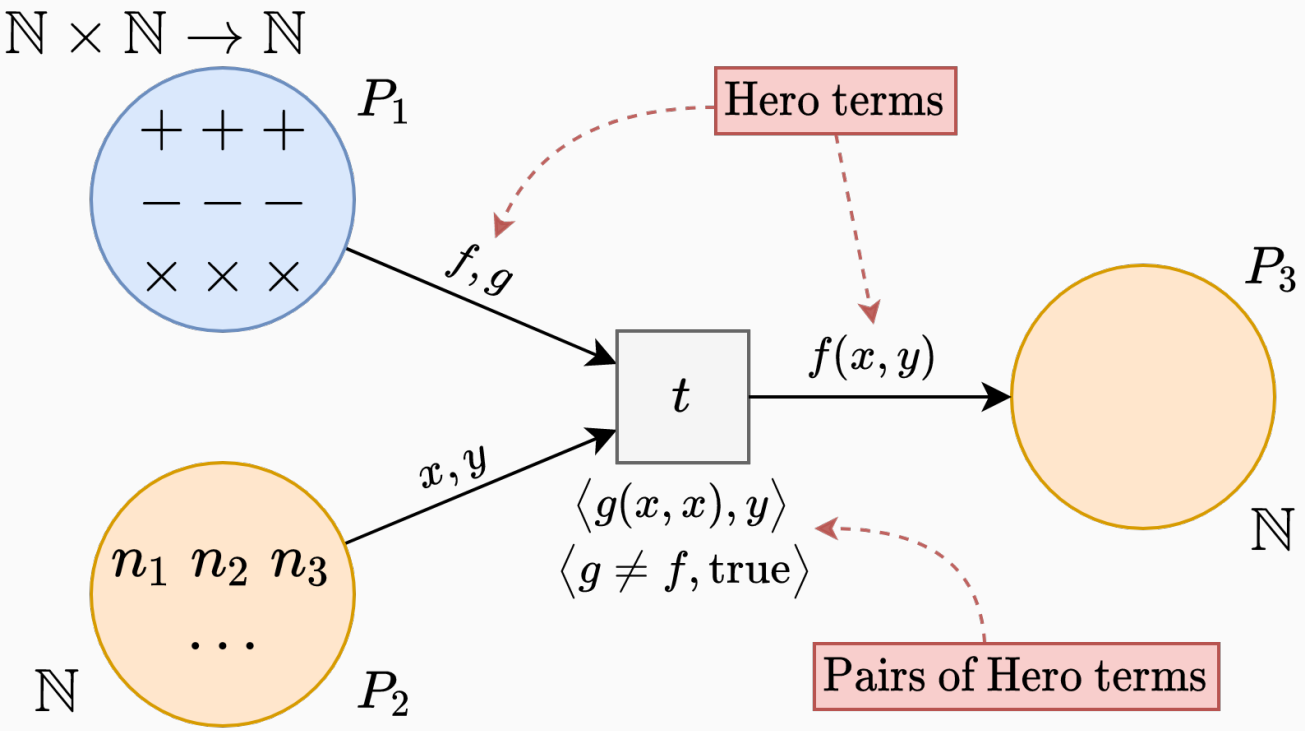
Hero net foundations

From Predicate nets...



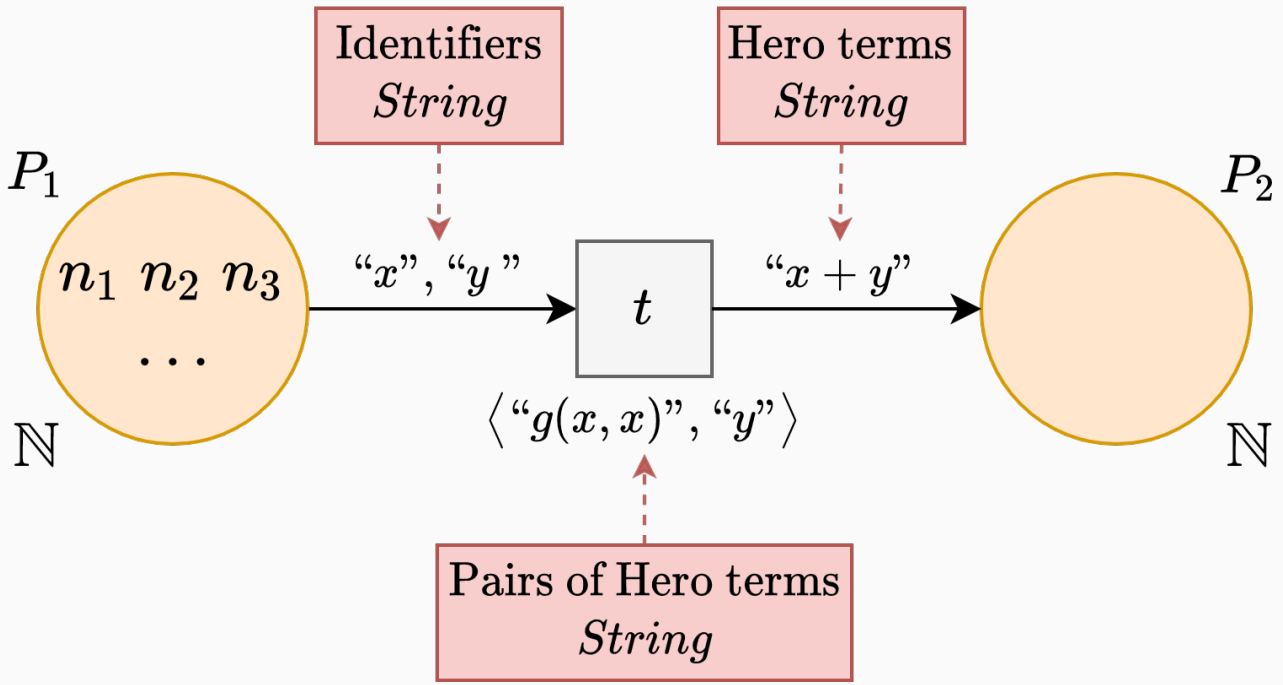
- How to model and execute **Hero nets**?
- Started with a library for *Predicate Nets*

... to Hero nets



- Token: set of values
- Transition arc: multiset of Hero terms
- Guard: set of pairs of Hero terms

Alpine integration with Hero nets



- Token: set of values (`Values`)
- Inbound arc: set of variable identifiers (`String`)
- Outbound arc: set of Hero terms (`String`)
- Guard: set of pairs of Hero terms (`String, String`)

Alpine interpreter modifications

- Input expression:
  - Ex: `"f(x, g(y)) or y"`
  - Type: `String`
- Input binding:
  - Ex: {`"f" -> and, "g" -> not, "x" -> true, "y" -> false`}
  - Type: `[String: Value]`
- Output: evaluation of the expression after substitutions
  - Ex: `true`
  - Type: `Value`

Evaluation (1/2)

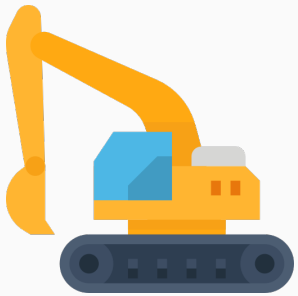
1. Parse the input expression into an untyped AST
    - `"f"`, `"g"`, `"x"` and `"y"` are `Ident` nodes
2. Substitute the `Ident` nodes with AST nodes extracted from values
3. Create the scopes and symbols to be associated with the AST nodes
4. Bind symbols to their respective scope
5. Find constraints over symbol types and solve them
6. Evaluate the typed AST with the corresponding evaluation contexts

Evalution (2/2)

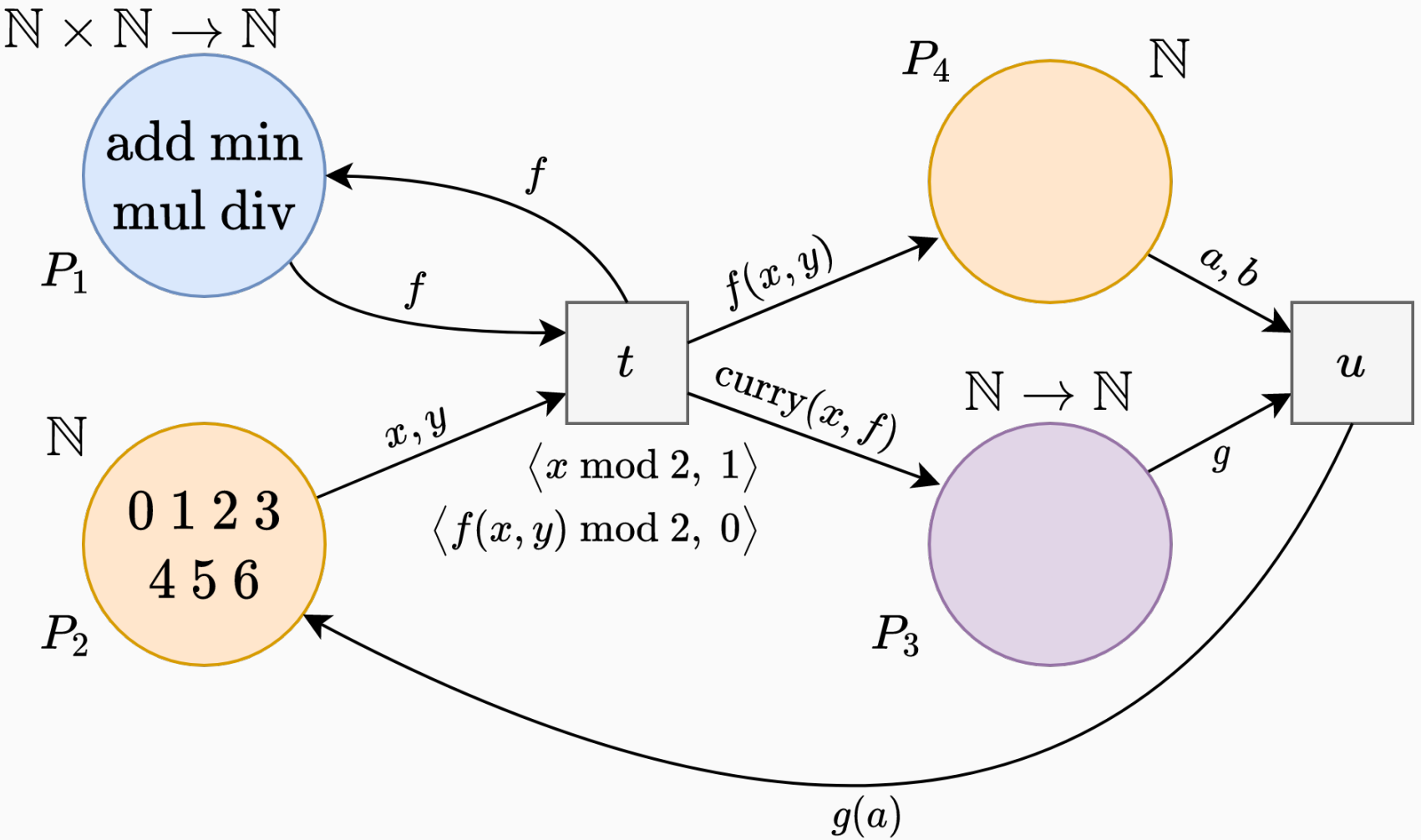
- `Value`'s type can be a:
  - Built-in one: `Bool`, `Int`, `Real`, `String`, `([Any]) -> Any`
  - Tuple: `(Tuple, [value: Value])`
  - Func: `(Func, EvaluationContext)`
- `Tuple` and `Func` are **AST nodes**, associated with a **module**, **symbols** and **scopes**

Difficulties

- Little to no experience with Swift
- At first no clear understanding of the different sub-steps
- Explored up some bad leads (e.g. making `Scope`, `Symbol` and every AST nodes conform to `NSCopying`)
- The evaluation context maps symbols to values, while the `runSema()` part works with scopes and symbols
- The module associated with scopes and every AST nodes was being deinitialized after the evaluation

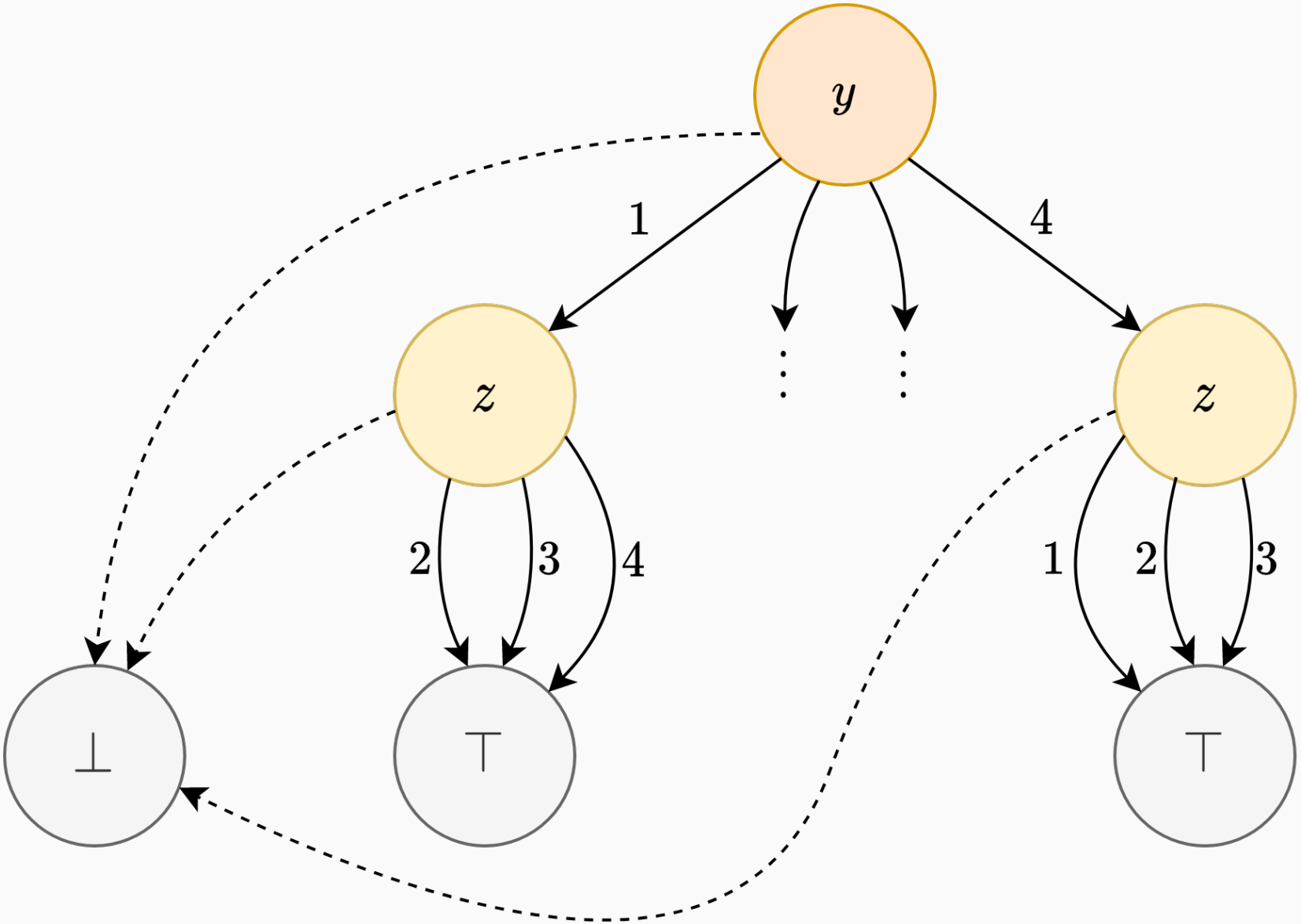


Operational Hero net interpreter

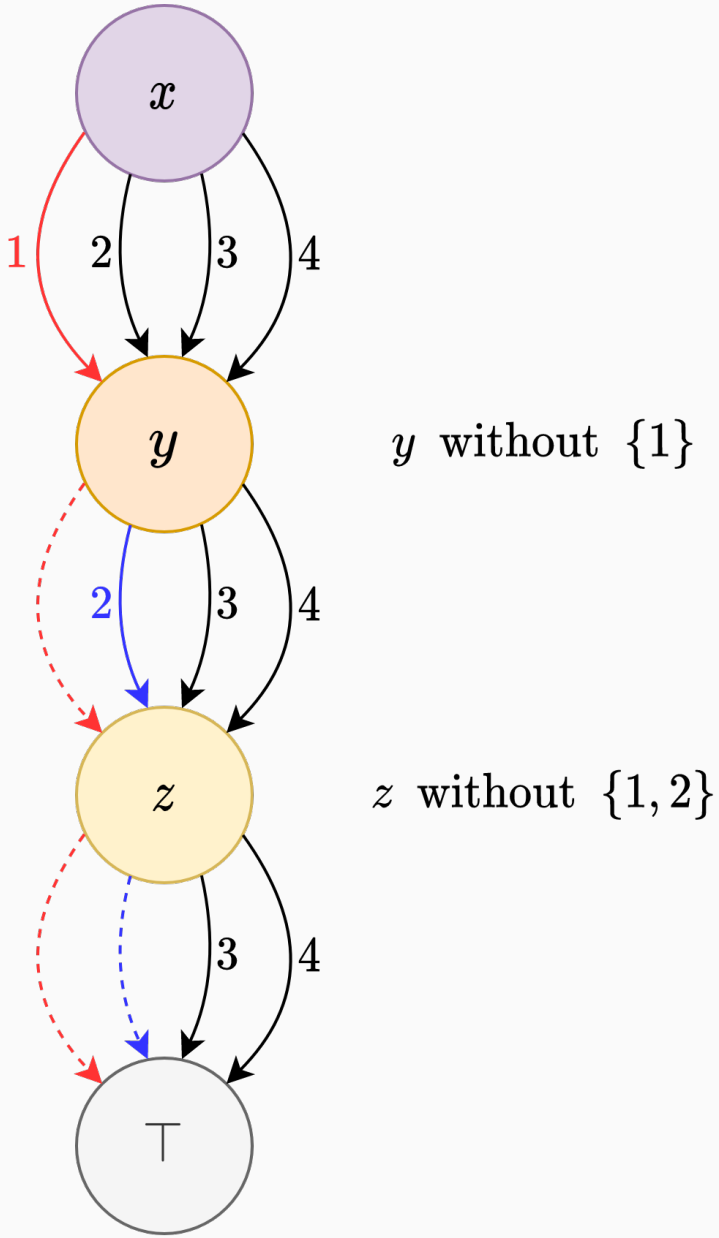


Efficient Hero net interpreter

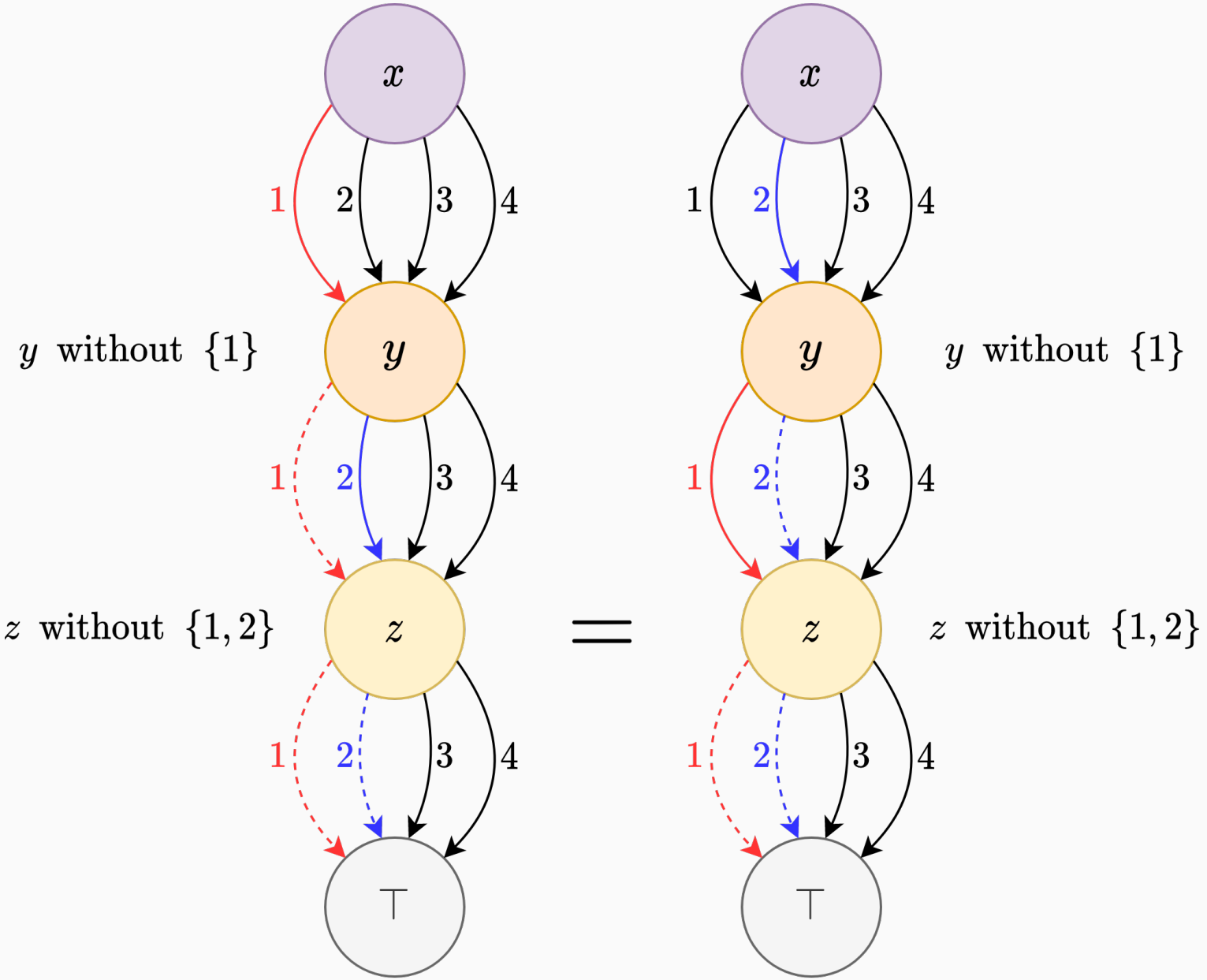
MFDD bindings



MFDD bindings: new method (1/2)



MFDD bindings: new method (2/2)



MFDD: fusion

Algorithm 1: Combining MFDDs representing permutations

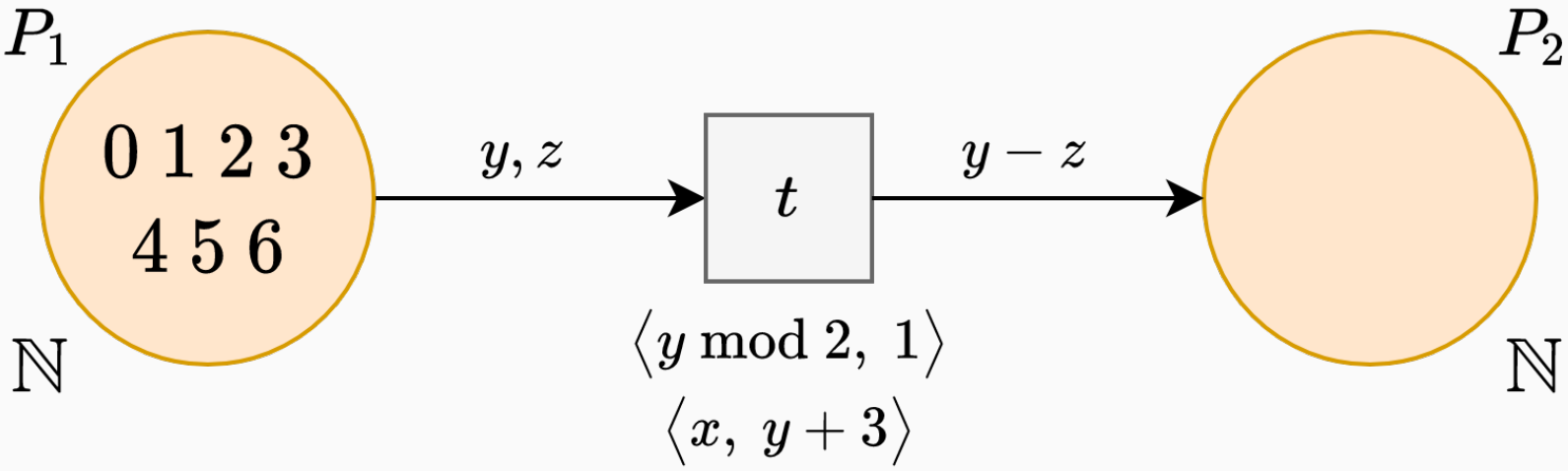
Input : lhs, rhs

Output: Fusion of lhs and rhs

Function `fusion(`*lhs*, *rhs*`)`**:**

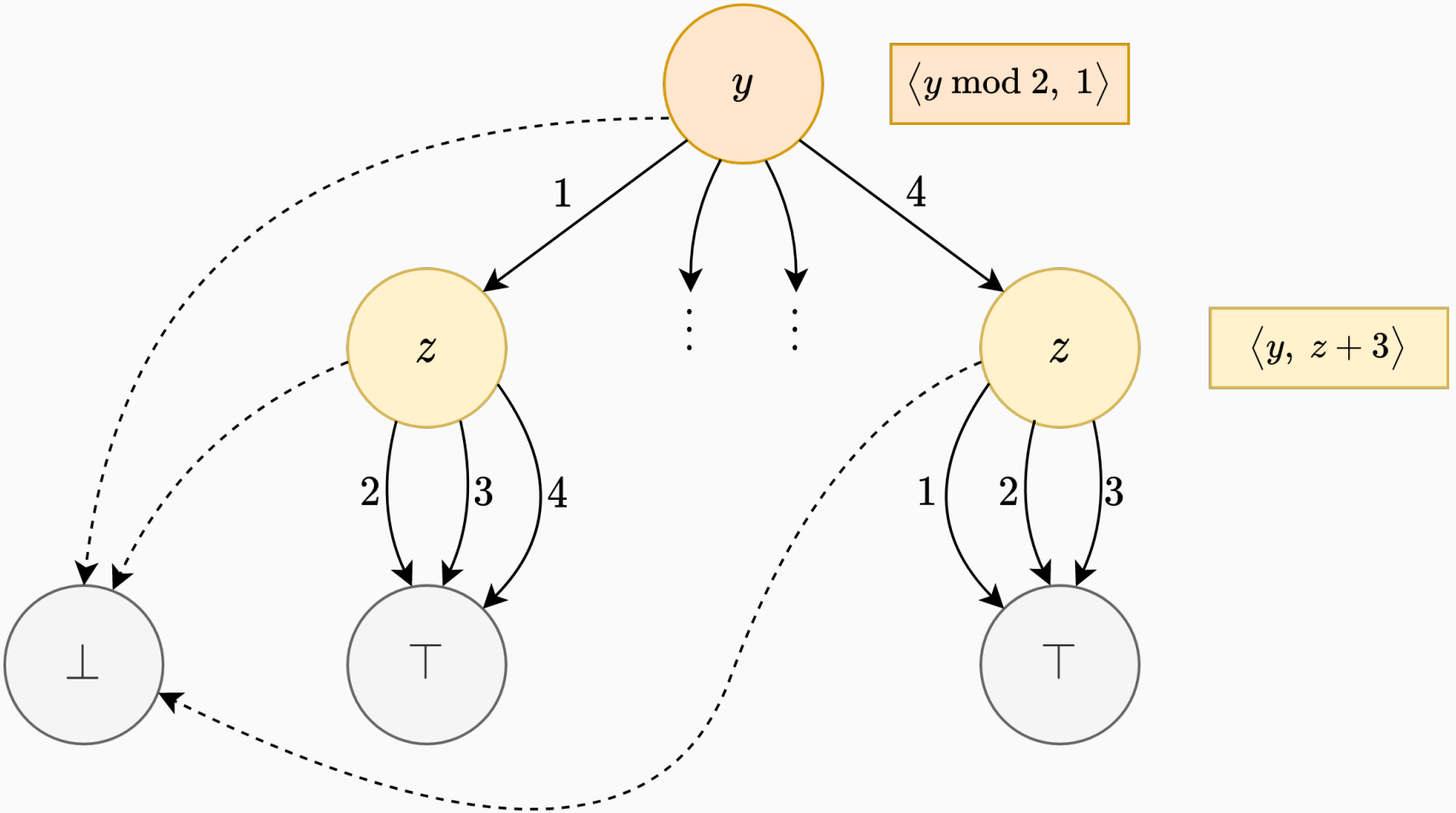
```
if lhs = one then
    return rhs
else if rhs = one then
    return lhs
if lhs.key > rhs.key then
    return fusion(rhs, lhs)
return node(
    lhs.key,
    lhs.take.mapValue { fusion($0, rhs) }
    zero)
```

Guards: simple strategy



1. Explore the MFDD until we find all the needed variables
2. Evaluate the two expressions with the resulting binding
3. Prune node if not equal

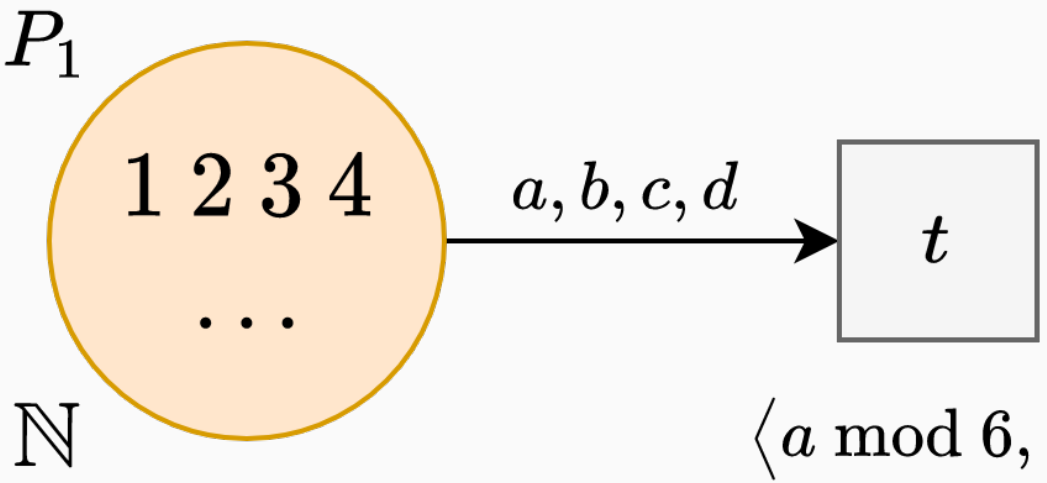
Guards: example



Several improvements

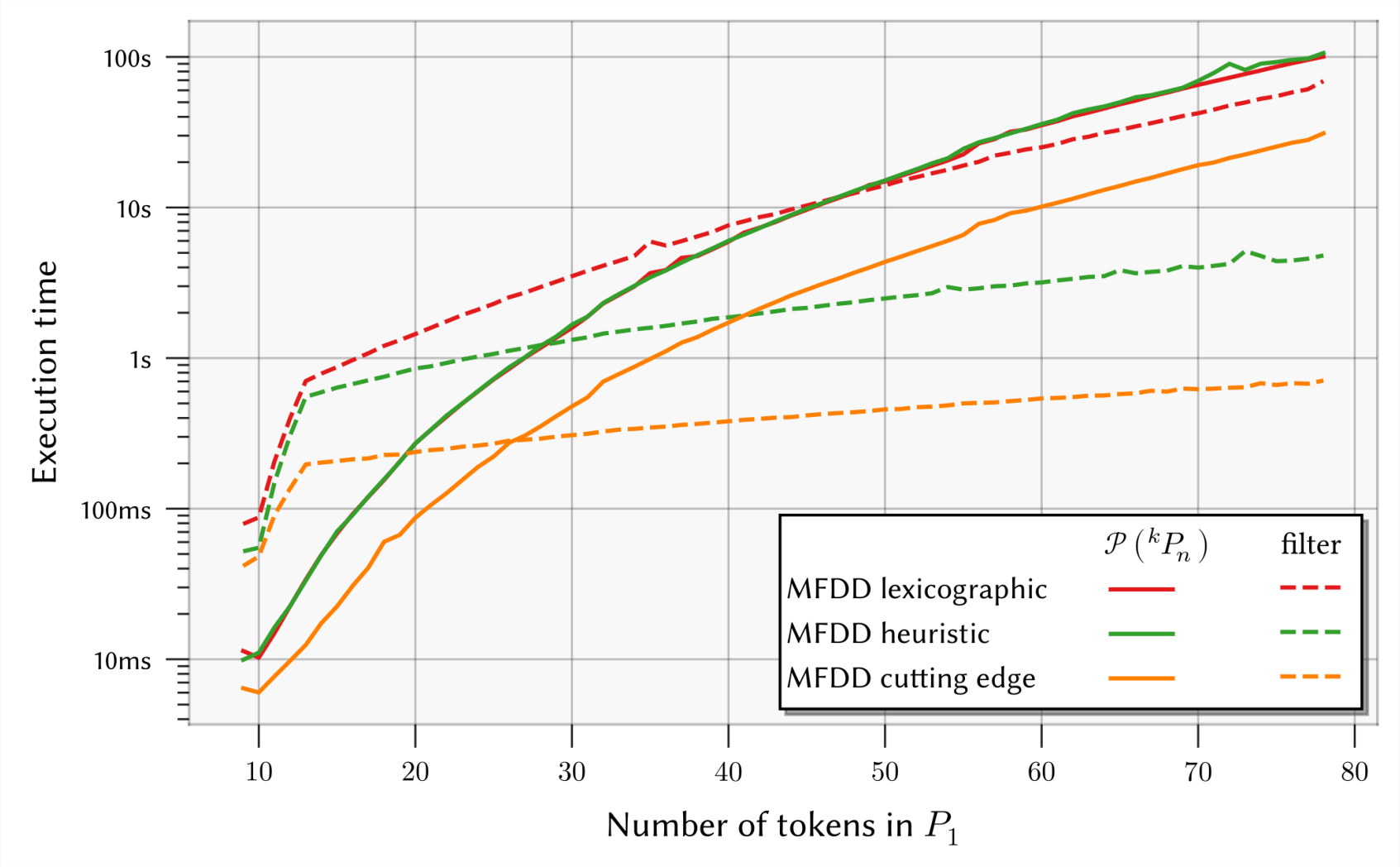
- Add a cache mechanism to
  - Fusion
  - Guard filtering
  - Alpine evalution
- Variable and guard ordering
  - Variables appearing in guards are put at the top
  - Of these variables, those in the guards with the smallest number of different variables have the smallest index
  - Guards featuring the fewest number of variables with the smallest indexes are checked first

Benchmark: lots of conditions (1/2)

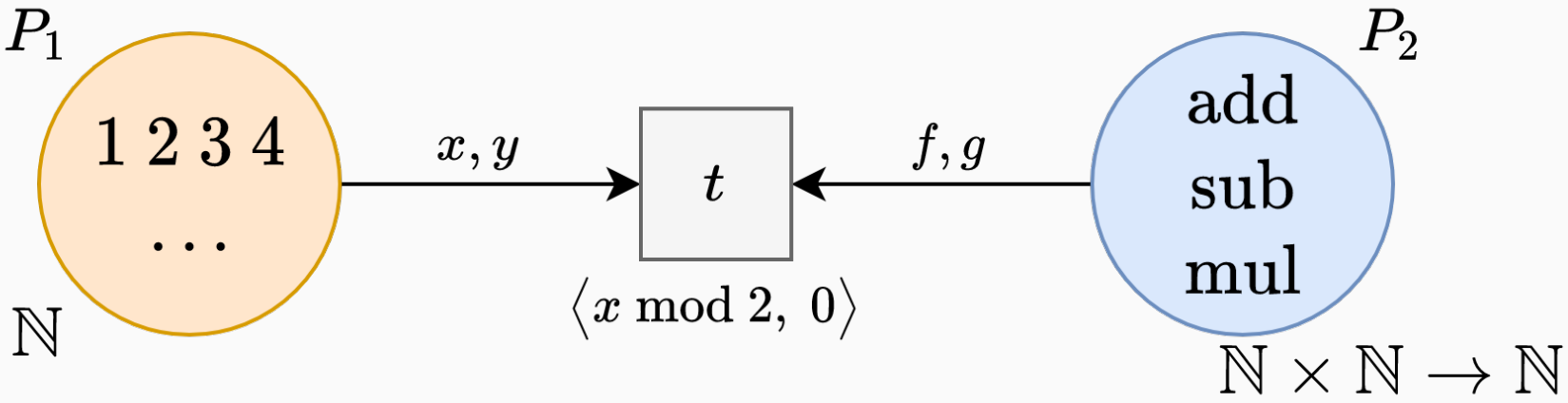


$$\langle a \bmod 6,\ b \rangle$$
$$\langle b < 4,\ \text{true} \rangle$$
$$\langle c > b \times 2,\ \text{true} \rangle$$
$$\langle d + b < a \times 2,\ \text{true} \rangle$$
$$\langle c < 10,\ \text{true} \rangle$$
$$\langle a < 10,\ \text{true} \rangle$$
$$\langle d < 10,\ \text{true} \rangle$$

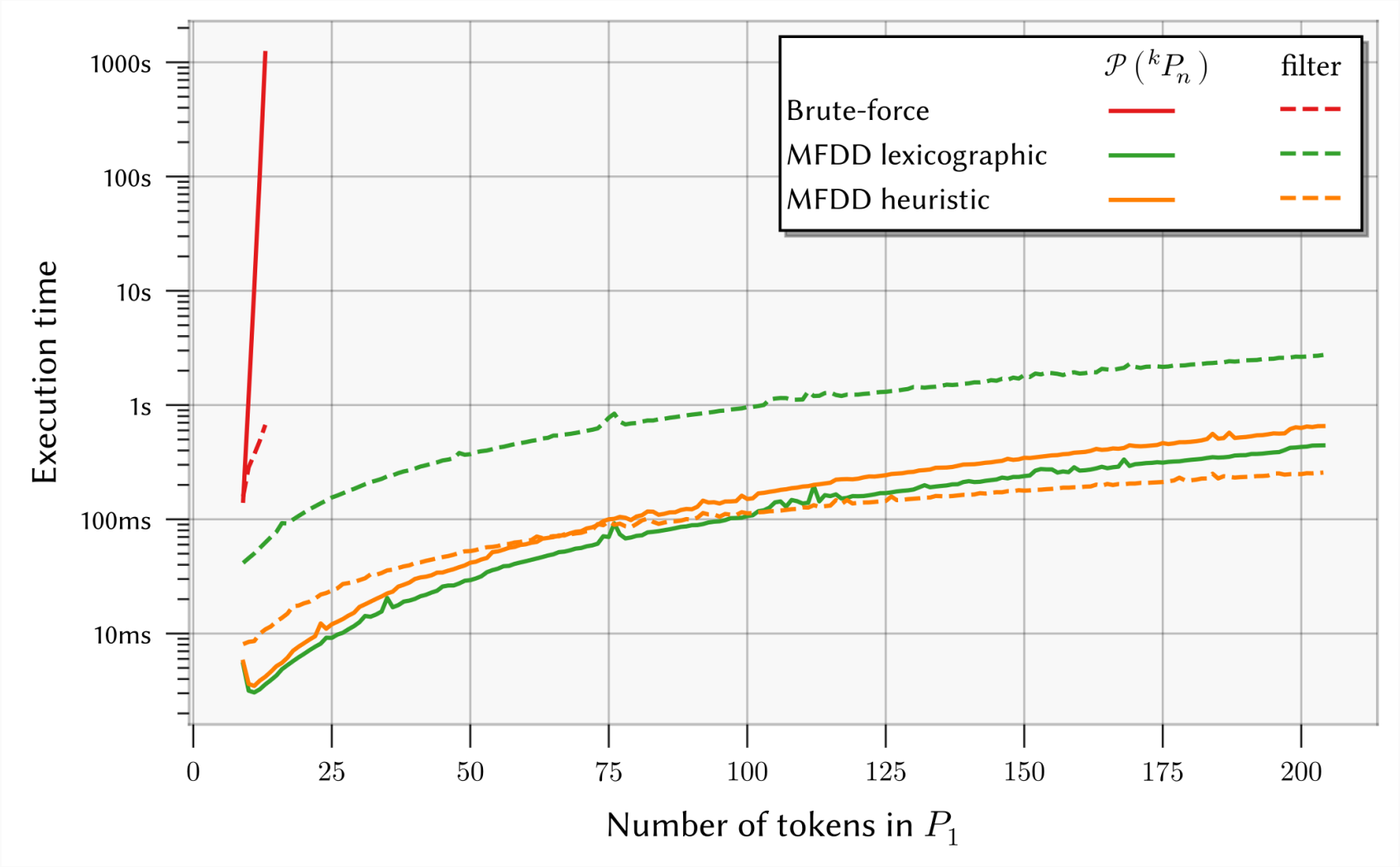
Benchmark: lots of conditions (2/2)



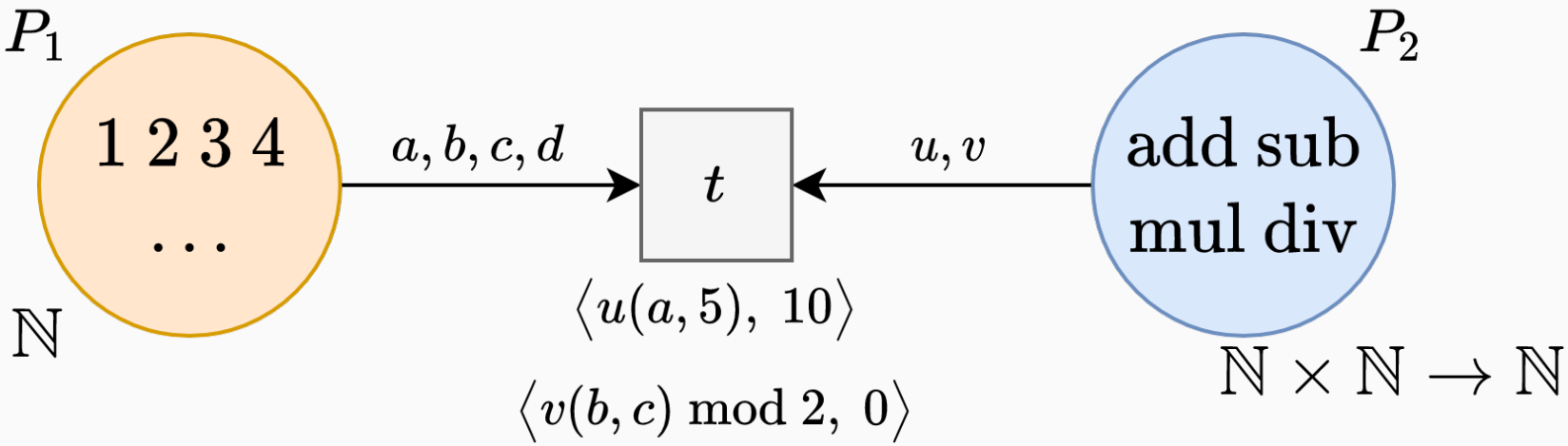
Benchmark: simple Hero net (1/2)



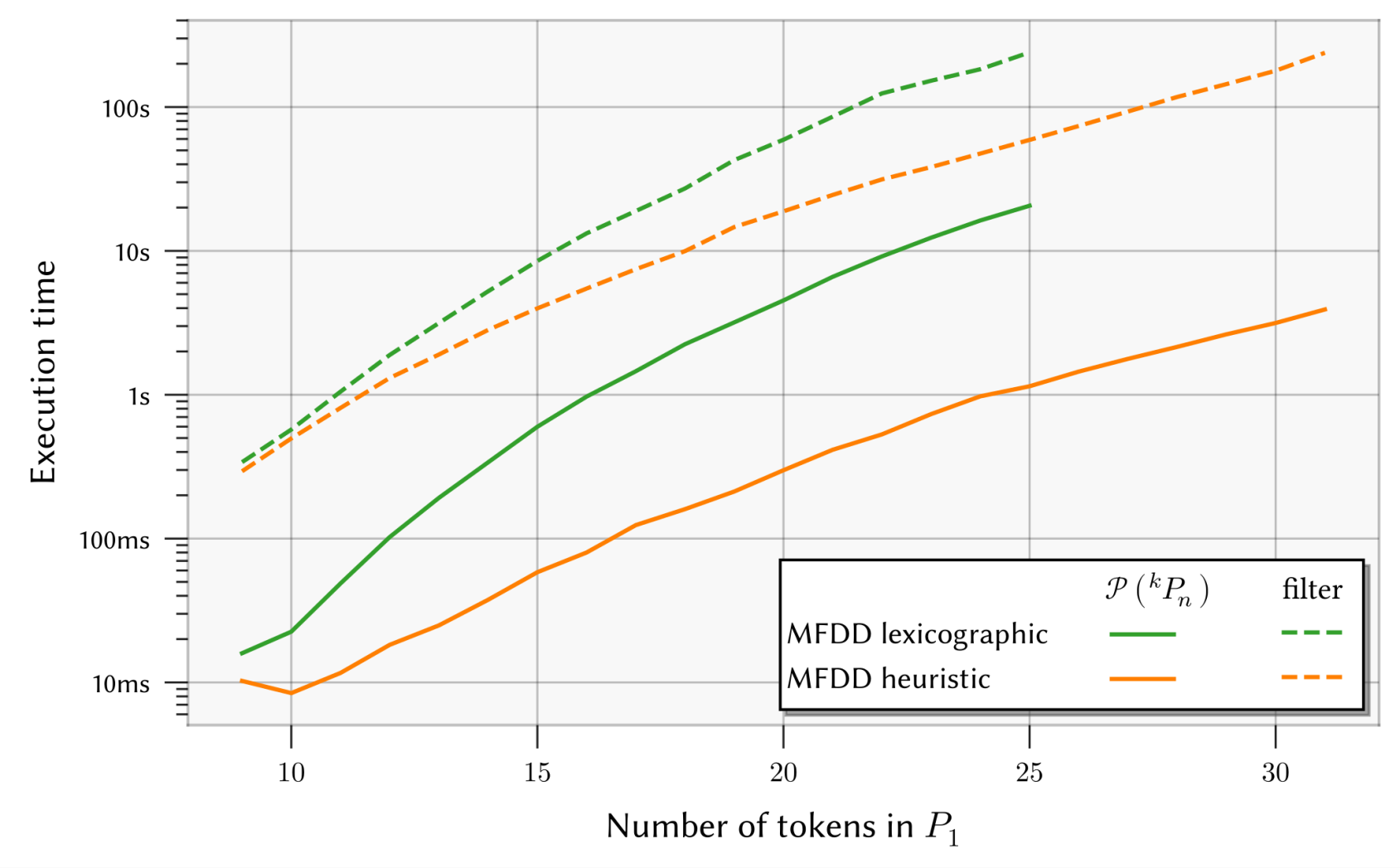
Benchmark: simple Hero net (2/2)



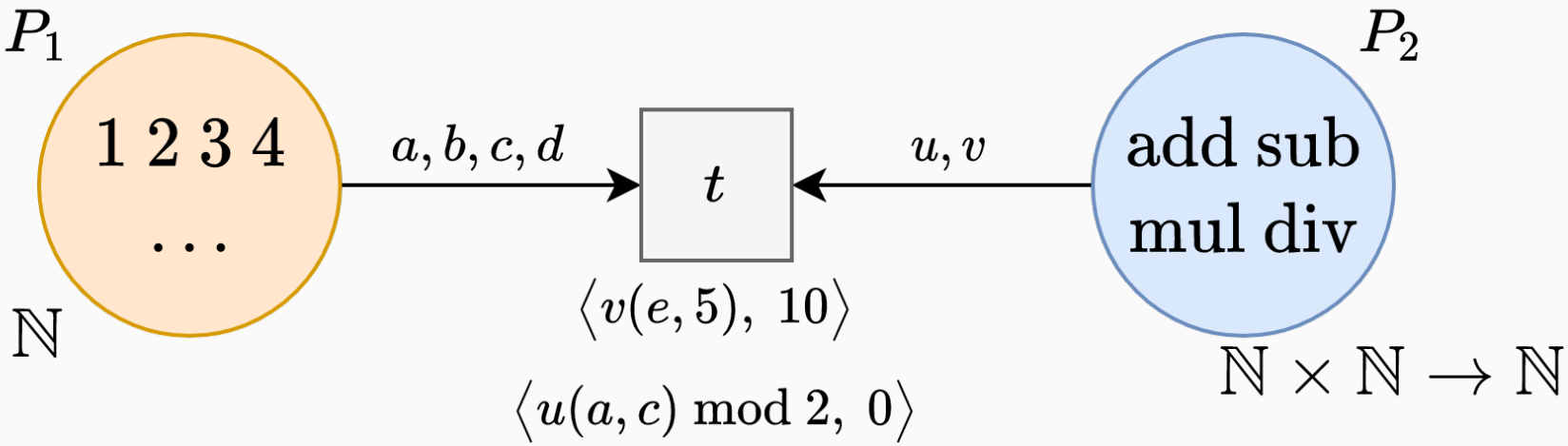
Benchmark: in order (1/2)



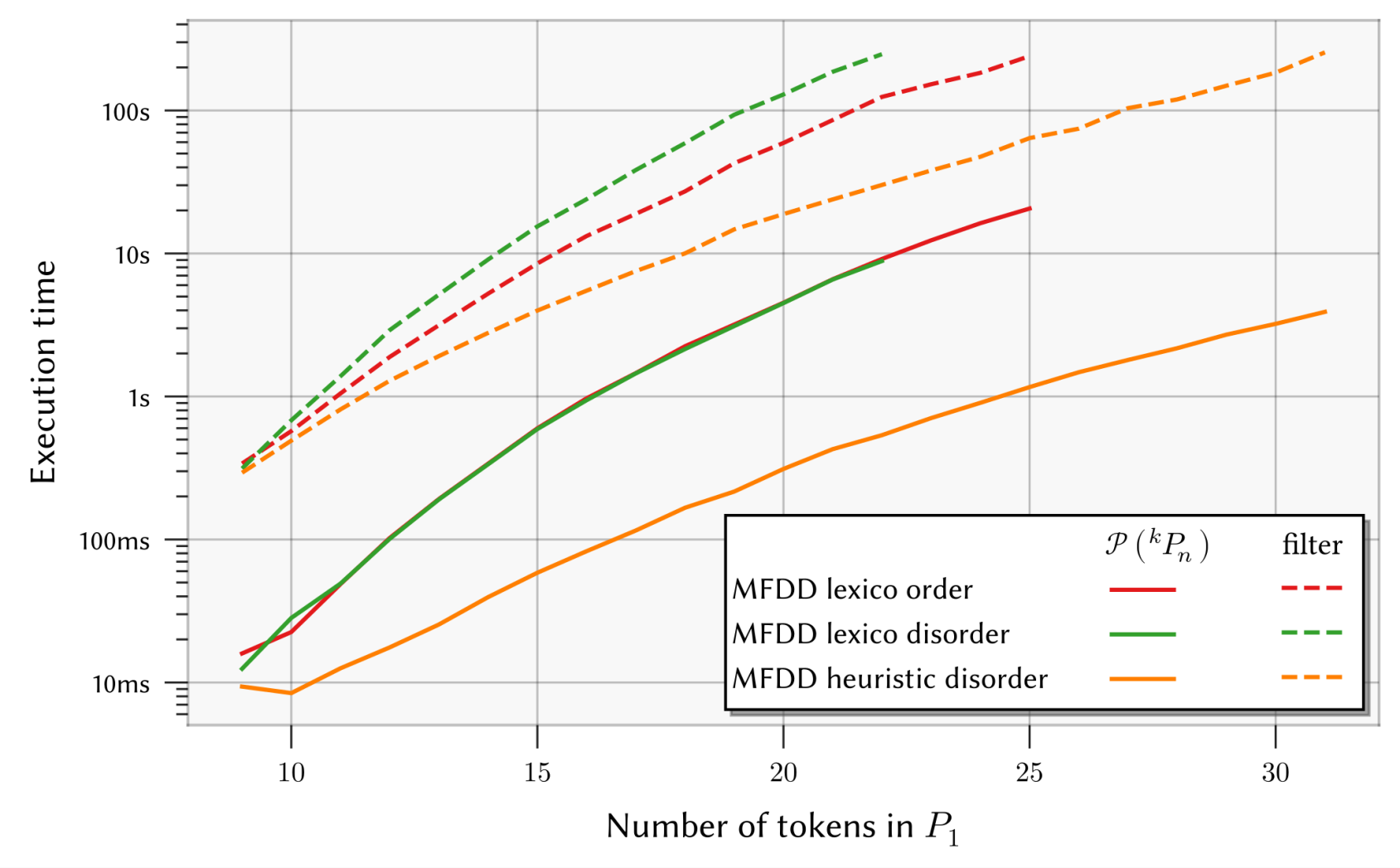
Benchmark: in order (2/2)



Benchmark: in disorder (1/2)



Benchmark: in disorder (2/2)



There is room for improvement!

- Clean the code
- Better caching mechanisms (especially with Alpine)
- Better variable ordering
- Better guard filtering (parse ASTs)
- Combine MFDD initial construction and guard filtering

Thank you!