

Compositional Recursion Analysis for Certifying Parallelizability of First-Order Functional Programs

Neea Rusch¹ and Nathaniel Young²

¹ Uppsala University, Sweden

² Cornell University, USA

The 38th Symposium on Implementation and Application of Functional Languages

29 October 2026

Overview

Classic problem in parallel computation

Which problems can be solved substantially faster using many processors?



Our formulation How to identify *programs* that can compute in parallel?

- The key idea is to detect parallelizability via membership in the complexity class NC.
- Instead of abstract machines, we infer the membership criterion from syntax.
- The analysis accepts programs that satisfy the membership criterion.
- We have implemented a machine that performs this analysis in practice.

Theoretical foundations

Characterizing NC via programming languages

NC "Nick's Class"

The complexity class NC is the **class of efficiently parallelizable problems**.

A problem with input size n is in the complexity class NC if it can be solved in polylogarithmic time using a polynomial number of parallel processors.

The NC includes a hierarchy of problems: $NC^0 \subseteq NC^1 \subseteq \dots \subseteq NC^i \subseteq \dots \subseteq NC$
e.g., NC^0 : constant-depth, NC^1 : polynomial-size with bounded fan-in, etc.

Characterizations of NC in literature:

- Machine level: NC^1 – Barrington (1987) NC^k – Immerman-Barrington-Straubing (1990) ...
- Machine-independent: NC^1 – Leivant-Marion (1999) NC^k – Clote (1990), Marion-Péchaux (2008) ...

Cook, S. A. (1983). An overview of computational complexity. *Communications of the ACM*, 26(6), 400–408. <https://doi.org/10.1145/358141.358144>;
Pippenger, N. (1979). On simultaneous resource bounds. In (Ed.), *20th Annual Symposium on Foundations of Computer Science: 20th Annual Symposium on Foundations of Computer Science*. <https://doi.org/10.1109/sfcs.1979.29>

Our core theory: Marion-Péchaux (2008)

In: Theory and Applications of Models of Computation: Vol. 4978 (2008),
DOI [10.1007/978-3-540-79228-4_12](https://doi.org/10.1007/978-3-540-79228-4_12).

A Characterization of NC^k by First Order Functional Programs

Jean-Yves Marion and Romain Péchoux

Loria-INPL, École Nationale Supérieure des Mines de Nancy, B.P. 239, 54506
Vandœuvre-lès-Nancy Cedex, France, Jean-Yves.Marion@loria.fr
Romain.Pechoux@loria.fr

Abstract. This paper is part of a research on static analysis in order to predict program resources and belongs to the implicit computational complexity line of research. It presents intrinsic characterizations of the classes of functions, which are computable in NC^k , that is by a uniform, poly-logarithmic depth and polynomial size family of circuits, using first order functional programs. Our characterizations are new in terms of first order functional programming language and extend the characterization of NC^1 in [9]. These characterizations are obtained using a complexity measure, the sup-interpretation, which gives upper bounds on the size of computed values and captures a lot of program schemas.

Functional programs

Syntax. Given constructors $c, v \in \text{Cns}$, pattern $p \in (\text{Var}, \text{Cns})$, variable $x \in \text{Var}$, expression $e \in (\text{Var}, \text{Cns}, \text{Op}, \text{Fct})$, operator $\text{op} \in \text{Op}$, and function $f \in \text{Fct}$, we define a program to consist of a list of rules $\mathcal{R} \in r$ defined by the following grammar:

(Values) $v := c \mid c(v_1, \dots, v_n)$

(Patterns) $p := c \mid x \mid c(p_1, \dots, p_n)$

(Expressions) $e := c \mid x \mid c(e_1, \dots, e_n) \mid \text{op}(e_1, \dots, e_n) \mid f(e_1, \dots, e_n)$

(Rules) $r := f(p_1, \dots, p_n) \rightarrow e$

The *main function* is the first function symbol in the list of rules. The *arity* of a symbol is the number of arguments that it takes. Only disjoint and linear rule patterns allowed; programs are confluent.

Functional programs with call-by-value semantics

Domains Values, errors, and divergence: $\mathcal{V} = \{\text{Cns}\}$, $\mathcal{V}_E = \mathcal{V} \cup \{\text{error}\}$, $\mathcal{V}_I = \mathcal{V}_E \cup \{\perp\}$

Operators An operator op of arity n denotes a non-iterative primitive: $[[e]] : \mathcal{V}^n \rightarrow \mathcal{V}_E$.
Operators cannot hide recursion.

Evaluation For $e \in \{\text{Cns}, \text{Op}, \text{Fct}\}$: $[[e]] = v$ iff $e \xrightarrow[R]{*} v$, $v \in \mathcal{V}_E$
Evaluation may raise an error or not terminate.

The main function Computes a partial function $\varphi : \mathcal{V}^n \rightarrow \mathcal{V}_E$: $\varphi(\bar{u}) = v$ iff $[[e]] = v$.

Value assignments A partial assignment maps each symbol of arity n to a partial function $(\mathbb{R}^+)^n \rightarrow \mathbb{R}^+$ defined for all constructors and operators.

Fraternities

A fraternity is a meta-construct representing **recursion and its surrounding context**.

Informally: a call must be in the set of rules, recursive calls are mutually recursive, and context symbols must be strictly smaller.

Fraternity. Given a program P , an expression $e = C(g_1(\overline{e_1}), \dots, g_r(\overline{e_r}))$ is a fraternity activated by a function call $f(p_1, \dots, p_n)$ if

1. $f(p_1, \dots, p_n) \rightarrow e$ is a rule of P .
2. For every $i \in \{1, \dots, r\}$, $g_i \approx_{\text{Fn}} f$.
3. For every symbol b in the context C , $f \approx_{\text{Fn}} b$.

Example: program P with 3 rules

```
f(x)      → rev(log(x))
log(i(x)) → if(msp(fh(i(x)), sh(i(x))),
              0(log(fh(i(x)))),
              1(log(fh(i(x)))))
log(ε)    → ε
```

The second rule, $\log(\dots) \rightarrow \log(\dots)$ is the only fraternity.

In 3.: constructors and operators are always smaller than functions.

Partial assignments and sup-interpretations

Partial assignments provide a mechanism to analyze computations.

sup-interpretation θ – a partial assignment that approximates data size.

- The main complexity measure; associated with symbols: Var, Cns, Op.
- Intuitively: bounds from above the output size of the program's function.

Some examples

$$\begin{array}{lll}
 \varepsilon : \text{Var} & i(x) : \text{Cns} & \text{if}(x, y) : \text{Op} \\
 \theta(\varepsilon) = 0 & \theta(i)(x) = x + 1 & \theta(\text{if})(x, y) = \max(x, y)
 \end{array}$$

lightweight ω – an approximation on *recursion depth* of function calls.

Arboreal programs

The arboreal criterion controls **recursion depth**. Arboreality ensures that the number of successive recursive calls is upper-bounded logarithmically in the input size.

Arboreality. A program P is arboreal iff there exist a polynomial and additive sup-interpretation θ , a monotonic polynomial partial assignment $\theta(c)$, and a constant $K > 1$ such that, for every fraternity $C(g_1(\overline{e_1}), \dots, g_r(\overline{e_r}))$ activated by $f(\overline{p})$, the following hold:

1. For every substitution $\mathcal{S}$, $\theta(c)(f)(\theta^*(\mathcal{S}(\overline{p}))) \geq 1$.
2. For every substitution $\mathcal{S}$ and every $i \in \{1, \dots, r\}$, $\omega(f)(\theta^*(\mathcal{S}(\overline{p}))) \geq K \times \omega(g_i)(\theta^*(\mathcal{S}(\overline{e_i})))$.
3. No function symbol h in $\overline{e_i}$ satisfies $h \underset{\text{Fn}}{\approx} f$.

Fraternal programs

The fraternal condition bounds **output size**. It ensures that the size of each computed value is polynomially bounded wrt. input size.

Fraternality. A program P is fraternal if there exists a polynomial and additive sup-interpretation θ such that, for each fraternity $C(g_1(\overline{e_1}), \dots, g_r(\overline{e_r}))$ activated by $f(p_1, \dots, p_n)$, and each symbol b of arity m appearing in context C or in some e_j , there exist constants $\alpha_{i,j}^b, \beta_j^b \in \mathbb{R}^+$ such that

$$\theta(b)(X_1, \dots, X_m) = \max_{j \in J} \left(\sum_{i=1}^m \alpha_{i,j}^b \times X_i + \beta_j^b \right)$$

where J is a finite index set and $\mathbb{R}^+ = [0, +\infty)$.

Important: *Do not confuse fraternity and fraternality.*

NC membership criterion

The characterization of arboreal and fraternal programs corresponds to functions computable in the complexity class NC.

Theorem. A function $\varphi: \mathcal{V}^n \rightarrow \mathcal{V}_E$ is computed by an arboreal and fraternal program if and only if φ is computable in NC.

$$\varphi \text{ computable by an arboreal and fraternal program} \iff \varphi \in \text{NC}$$

Initial questions

Automation How to convert the characterization to an algorithm?

Practicality Is the theory implementable, perhaps after some modification?

Expressivity what problems can be expressed within the criteria?

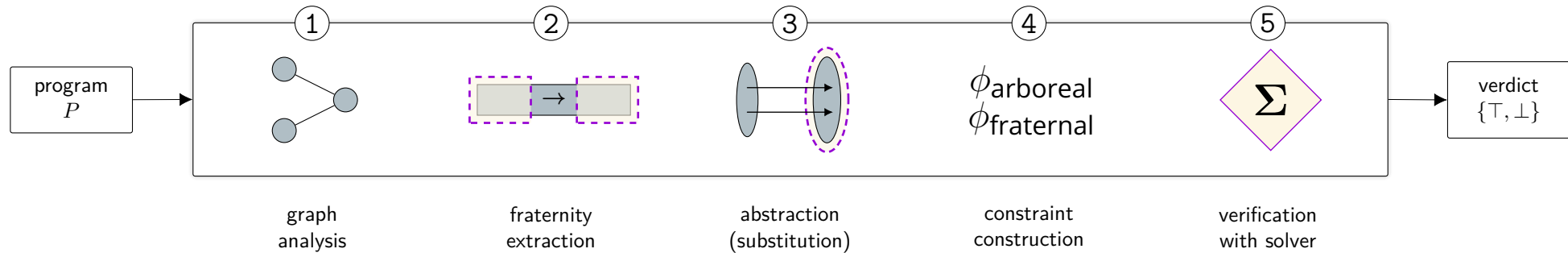
Insights What are the difficult steps, e.g, which complexity criterion is easier?

Transforming the core theory into

Automatable program analysis

Program analysis overview

Based on the theory of arboreal and fraternal programs, we have designed an automatic program analysis. A positive verdict certifies membership in class NC.



1. Graph analysis

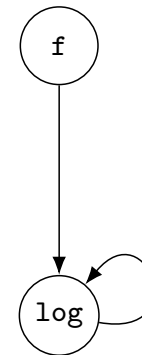
Generate graphs that carry information about recursive calls and data dependence between rules.

Essentially, our call graph is a directed graph where:

- Vertices represent the rules in $\mathcal{R}$.
- There is an edge between vertices f and g iff there is a function call from f to g in $\mathcal{R}$.

From the call graph, we can compute its transitive closure and strongly connected components.

```
f(x)      → rev(log(x))
log(i(x)) → if(msp(fh(i(x)), sh(i(x))),
              0(log(fh(i(x)))),
              1(log(fh(i(x))))))
log(ε)     → ε
```



2. Fraternity extraction

From the list of rules and with graphs, we derive fraternities.

Recall, we look for:

1. maximal recursion pattern
2. at same level of recursion
3. with strictly smaller context symbols

Essentially, we extract from the input program, the information that contributes to the complexity measurement.

```
f(x)      → rev(log(x))
log(i(x)) → if(msp(fh(i(x)), sh(i(x))),
              0(log(fh(i(x)))),
              1(log(fh(i(x)))))
log(ε)    → ε
```

The example program has 1 fraternity:

```
function: log
patterns: i(x)
context:  if(msp(fh(i(x)), sh(i(x))), 0(◇), 1(◇))
calls:    log
```

3. Abstraction with sup-interpretations

Infer sub-interpretations and lightweights for symbols in fraternities.

Important sup-interpretation bounds *from above* the *output size* of a symbol.

- Complexity measurement is on sup-interpretations instead of “raw” symbols.
- Abstraction that overestimates the program’s true runtime behavior.
- Gives us evaluable numerical expressions to use with a solver.

In the generic first-order language:

- Treatment of variables, constructors, and calls is straightforward.
- Operators cannot be inferred directly → required algorithmic input.

$$\begin{array}{lll}
 \varepsilon : \text{Var} & i(x) : \text{Cns} & \text{if}(x, y) : \text{Op} \\
 \theta(\varepsilon) = 0 & \theta(i)(x) = x + 1 & \theta(\text{if})(x, y) = \max(x, y)
 \end{array}$$

4. Constraint construction

From the fraternities and symbolic abstractions: construct verification conditions.

Arboreality

- Goal: Establish that the program cannot violate logarithmic recursion depth.
- *Proof by unsatisfiability*: the solver's task is to find a violating counterexample.

Fraternity

- Goal: infer upper bounds on output sizes of symbols.
- *Proof by bounding*: Show that there exists an upper bound of acceptable shape.
⇒ the symbol output size is also necessarily acceptable.

Parallelizability question is now reduced to a solver-evaluatable problem.

5. Verification

Evaluate arboreality and fraternality with SMT solver.

The analysis verdict is positive if both criteria pass the verifier.

Observations:

- Complexity criteria is independent – can evaluate in parallel.
- The evaluation effort varies – arboreality is easier to solve.
- Fraternity verification could be lifted by restricting the input language.
- Fraternity must bound each symbol once, regardless of number of uses.

Evaluation

We have developed an analyzer that implements the analysis. The analyzer also tracks meta-information about the analysis, e.g., constraint solutions and program statistics.

- Benchmarks: some programs acceptable, others not in NC.
- Two backend solvers: Z3 and cvc5

Observations:

- Analysis is lightweight, but solver behavior is unpredictable.
- Both solvers evaluate arboreality successfully and in agreement.
- Z3 succeeds at fraternity constraints, but cvc5 mostly times out.
- It is beneficial to decompose the large conjunctive constraints.

Demo analysis run

input

```
f(x)      → rev(log(x))
log(i(x)) → if(msp(fh(i(x)), sh(i(x))),
               0(log(fh(i(x)))),
               1(log(fh(i(x)))))
log(ε)    → ε
```

terminal

```
Program count ..... 1
Solver timeout, sec..... 1
Time measurements ..... 5

Solver: z3
Program      #R #F #S   A F V   t_ms ±σ
log          3  1  7   ✓ ✓ ✓  12.69 ±1.27
Unknown or reject ..... 0
Accepted ..... 1
```

result (clipped)

```
n_rules: 3
n_fraternalities: 1
n_symbols: 7
error: null
arboreal:
  arbo_coef: 2
  verdict: true
  unknown: null
  cex: null
  conjecture: |-
    [x >= 0,
     Or(0 + 0 + x + 1 < 1,
        0 + 0 + x + 1 < 2*(0 + (0 + x + 1)/2),
        0 + 0 + x + 1 < 2*(0 + (0 + x + 1)/2))]
fraternal:
  verdict: true
  unknown: false
  n_symbols: 7
  results:
    if:
      poly: true
      sol:
        bound_exp: Theta(if)(X1,X2,X3) <= a1*X1 + a2*X2 + a3*X3 + B
        values: '[B = 0, a2 = 2, a1 = 0, a3 = 2]'
        simplified: 2*X2 + 2*X3
    :
```

Summary

It works! the NC characterization extends to an automatable program analysis of parallelizability.

Summary & remaining questions

It works! the NC characterization extends to an automatable program analysis of parallelizability.

- **Appropriate formulation of completeness**

Established previously: a function in NC can be simulated by an arboreal and fraternal program, using divide-and-conquer algorithms, but we do not know the program.

- **Motivating a meaningful evaluation**

The interesting question is how to find the right program, while the analyzer plays an affirmative role.

- **Increasing verification rigor with formal methods**

Our algorithmic design relies on the fact that we have accurately interpreted the core theory. Although the final verdict comes from a solver, we assume our algorithm constructs the correct constraints.