Recursion Analysis for Certifying Parallelizability

Neea Rusch[1] and Nathaniel Young[2]

[1] Uppsala University, Sweden
[2] Cornell University, USA

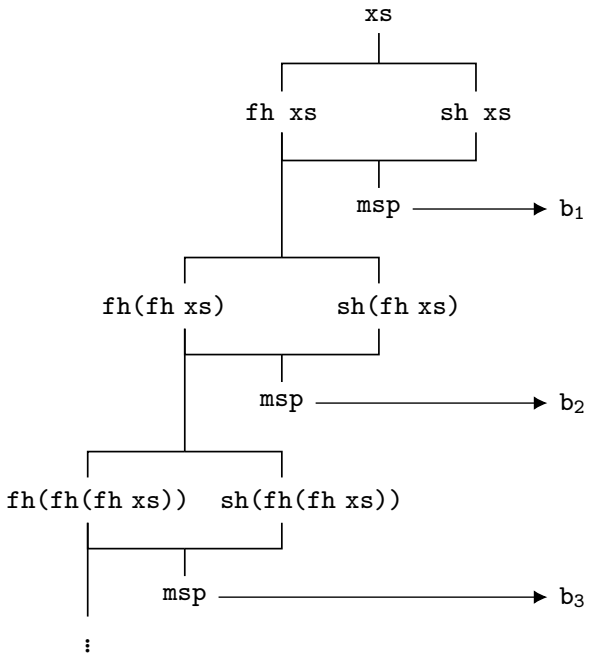
36th Nordic Workshop on Programming Theory
20 November 2026

```
let rec log_fun = function
  | [] -> []
  | xs -> (if msp (fh xs) (sh xs)
           then 0 else 1) :: log_fun (fh xs)
```

`xs` is a bitvector, `fh` first half of `xs`, `sh` second half of `xs`, and `msp` is a kind of length parity checker.
Program computes the strict ceiling logartihm, $\mathrm{sclog}_2(n) = \min\{k \in N : n < 2^k\}$, where `xs` encodes n as bits.

```
let rec log_fun = function
  | [] -> []
  | xs -> (if msp (fh xs) (sh xs)
           then 0 else 1) :: log_fun (fh xs)
```

Recursive unfolding



```
let rec log_fun = function
  | [] -> []
  | xs -> (if msp (fh xs) (sh xs)
           then 0 else 1) :: log_fun (fh xs)
```

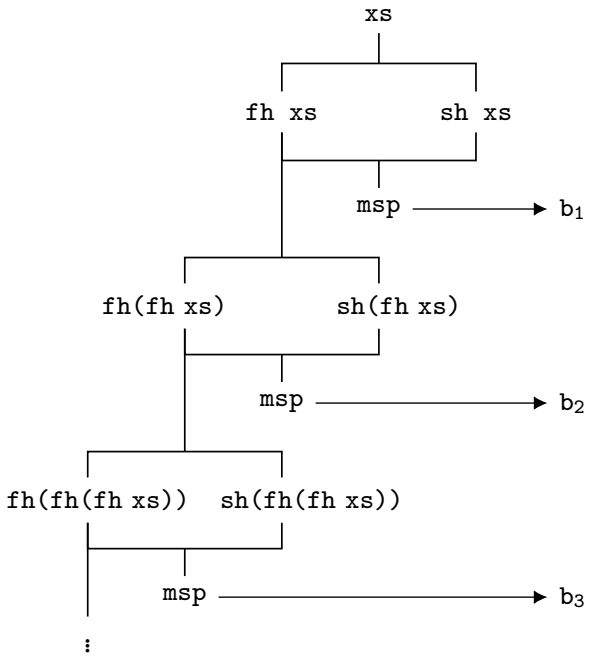
```
              fh xs = 10110111

          fh (fh xs) = 1011

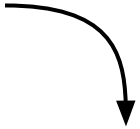
      fh (fh (fh xs)) = 10

fh (fh (fh (fh xs))) = 1
```

Recursive unfolding



```
let rec log_fun = function
  | [] -> []
  | xs -> (if msp (fh xs) (sh xs)
           then 0 else 1) :: log_fun (fh xs)
```



parallel variant

```
let log_fun =
Array.map Domain.join
(Array.init n (fun i ->
  Domain.spawn (fun () ->
    let p = prefixes.(i) in
    if msp (fh p) (sh p) then 0 else 1
)))
```

```
fh xs = 10110111

fh (fh xs) = 1011

fh (fh (fh xs)) = 10

fh (fh (fh (fh xs))) = 1
```

Main ideas

Question

How to determine automatically if a recursive first-order functional programs could compute its output in parallel?

Method overview

detect parallelizability via membership in complexity class NC

instead of abstract machines, infer NC membership from syntax

accept or reject programs by whether they satisfy the NC criteria

we have built a machine that performs the analysis in practice

Theoretical foundations

Characterizing NC via programming languages

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: $\mathrm{NC}^0 \subseteq \mathrm{NC}^1 \subseteq \cdots \subseteq \mathrm{NC}^i \subseteq \cdots \subseteq \mathrm{NC}$.

Machine-level characterizations

NC^1	Barrington (1987)
NC^k	Immerman-Barrington-Straubing (1990)
...	

Machine-independent

NC^1	Leivant-Marion (1999)
NC^k	Clote (1990)
NC^k	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

In: Theory and Applications of Models of Computation
Vol. 4978 (2008), DOI 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.

First-order functional programs

A program consists of a list of rules $\mathcal{R} \in r$ defined by the following grammar.
The *main function* is the first function symbol in the list of rules.

variable $x \in \mathrm{Var}$
constructors $c, v \in \mathrm{Cns}$
operator $\mathtt{op} \in \mathrm{Op}$
function $f \in \mathrm{Fct}$
pattern $p \in (\mathrm{Var}, \mathrm{Cns})$
expression $e \in (\mathrm{Var}, \mathrm{Cns}, \mathrm{Op}, \mathrm{Fct})$

$$
\begin{aligned}
\textit{(Values)}\quad & v &&:= c \mid c(v_1, ..., v_n) \\
\textit{(Patterns)}\quad & p &&:= c \mid x \mid c(p_1, ..., p_n) \\
\textit{(Expressions)}\quad & e &&:= c \mid x \mid c(e_1, ..., e_n) \mid \\
& && \quad \mathtt{op}(e_1, ..., e_n) \mid f(e_1, ..., e_n) \\
\textit{(Rules)}\quad & r &&:= f(p_1, ..., p_n) \to e
\end{aligned}
$$

```
let rec log_fun = function
  | [] -> []
  | xs -> (if msp (fh xs) (sh xs)
           then 0 else 1) :: log_fun (fh xs)
```



```
f(x)      → 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
```

Functional programs with call by-value-semantics

```
f(x)      →  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
```

Arity is the number of arguments that a symbol (Var, Cns, Op, Fct) takes.

Operator of arity n is a non-iterative primitive.

Evaluation may raise an error or diverge.
Domains: $\mathcal{V} = \{\mathrm{Cns}\}, \quad \mathcal{V}_E = \mathcal{V} \cup \{\texttt{error}\}, \quad \mathcal{V}_I = \mathcal{V}_E \cup \{\bot\}$

A *partial assignment* maps each symbol of arity n to a partial function $(\mathbb{R}^+)^n \to \mathbb{R}^+$.

Fraternities

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

Informally: a call must be in the set of rules rules, recursive calls are mutually recursive (at same level or recursion), and all context symbols must be strictly smaller. [2]

The example is a program of 3 rules.

```
1  f(x)      →  log(x)
2  log(i(x)) →  if(msp(fh(i(x)), sh(i(x))),
3                0(log(fh(i(x)))),
4                1(log(fh(i(x)))))
5  log(ε)    →  ε
```

The second rule `log(…)` $\rightarrow$ `log(…)`:

- the only recursive rule
- satisfies the other conditions

$\Rightarrow$ the only fraternity

[2] Constructors and operators are always strictly smaller than functions.

Sup-interpretations

Partial assignments provide a mechanism to analyze computations.

Sup-interpretation θ is a partial assignment that approximates data size.

- The main complexity measure associated with symbols $\mathrm{Var}, \mathrm{Cns}$, and Op.
- Intuitively, it bounds from above the output size of the program's function.

```
f(x)      →  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
```

type	symbol	sup-interpretation
$\texttt{ε} : \mathrm{Var}$	ε	$\theta(\varepsilon) = 0$
$\texttt{c(…)} : \mathrm{Cns}$	i(x)	$\theta(\mathrm{i})(x) = x + 1$
$\texttt{fh(…)} : \mathrm{Op}$	fh(x)	$\theta(\mathrm{fh})(x) = x \div 2$
$\texttt{f(…)} : \mathrm{Op}$	if(b,x,y)	$\theta(\mathrm{if})(b, x, y) = \max(x, y)$

,

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 $C\left(g_1(\overline{e_1}), ..., g_{r(\overline{e_r})}\right)$ 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 $i \in \{1, ..., r\}$, $\omega(f)(\theta^*(\mathcal{S}(\overline{p}))) \geq K \times \omega(g_i)(\theta^*(\mathcal{S}(\overline{e_i})))$.
3. No function symbol h in $\overline{e_i}$ satisfies $h \underset{\text{Fn}}{\approx} f$.

Informally: *the recursive measure must be non-zero, the recursive weight shrinks, and recursive calls cannot contain nested recursive calls.*

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 $C\left(g_1(\overline{e_1}), ..., g_{r(\overline{e_r})}\right)$ activated by $f(p_1, ..., p_n)$, and each symbol b of arity m appearing in context C or in some e_j, there exist constants $\alpha^b_{i,j}, \beta^b_j \in \mathbb{R}^+$ such that

$$\theta(b)(X_1, ..., X_m) = \max_{j \in J} \left(\sum_{i=1}^{m} \alpha^b_{i,j} \times X_i + \beta^b_j \right)$$

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

Informally: *in a fraternal program, every symbol in a context or in an argument of a fraternity admits a bounded sup-interpretation.*

NC membership criterion

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

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

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

Marion, J.-Y., & Péchoux, R. (2008). A Characterization of NC^k by First Order Functional Programs. In M. Agrawal, D. Du, Z. Duan, & A. Li (Eds.), *Theory and Applications of Models of Computation: Vol. 4978. Theory and Applications of Models of Computation*. https://doi.org/10.1007/978-3-540-79228-4_12

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 NC criteria?

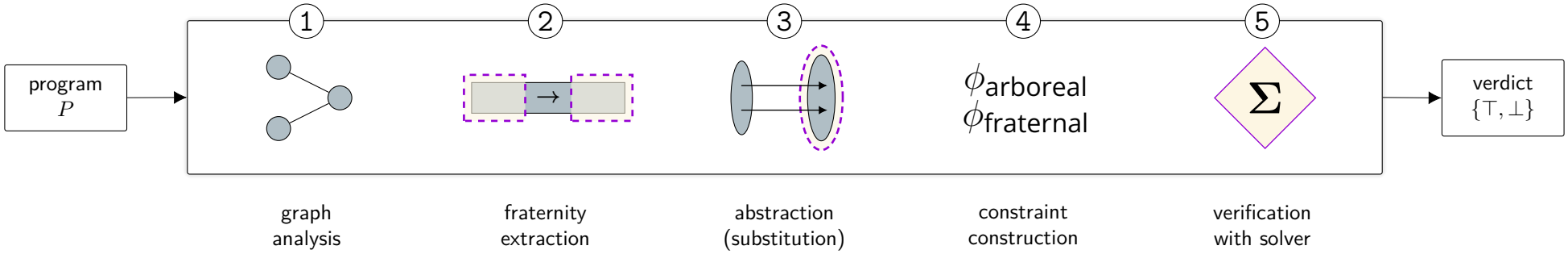
Insights What are the difficult steps; for example, which criterion is easier to check?

Transforming the core theory into

Automatable program analysis

Program analysis overview

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



1. Graph analysis

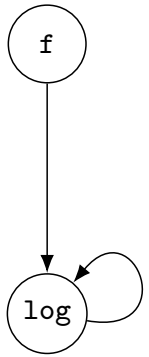
Graphs carry information about recursive calls and data dependence between rules.

Essentially, our call graph is a directed graph where

- vertices represent the rules in $\mathcal{R}$, and
- 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)      →  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 the graphs, we derive fraternities.

We look for:

1. maximal recursion
2. at the 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)      →  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
```

Recall, 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 symbols in fraternities.
Important: sup-intererpretation bounds *from above* the *output size* of a symbol.

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

The symbols in context are:
`i(x)`, `if(…)`, `msp(…)`, `fh(…)`, `sh(…)`, `0(…)` and `1(…)`

Variables, constructors, and calls are easy to infer
Operators cannot be inferred directly

This is an abstraction step that overestimates program's true behavior. It gives us evaluatable numerical expressions to use with a solver.

Complexity measurement is performed over sup-interpretations instead of "raw" symbols.

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.

Fraternality

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

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

Demo

input

```
f(x)      →  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
```

results [fragment]

```yaml
n_rules: 3
n_fraternities: 1
n_symbols: 7
error: null
arboreal:
  arbo_coef: 2
  verdict: true
  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
  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
    ⋮
```

5. Verification

Evaluate arboreality and fraternality with an SMT solver. The verdict is positive if both criteria pass the verifier.

Observations (i.e., what we know now):

- Complexity criteria are independent – can be evaluated in parallel.
- Fraternality verification could be lifted by restricting the input language.
- Fraternality must bound each symbol once, regardless of uses.
- The evaluation effort varies – arboreality is easier to solve.
- Solver performance is not uniform – by our experiments, Z3 works best.

Outstanding unknowns: how to formulate completeness, whether to formalize the analysis, and how to motivate meaningful evaluation.