The Complexity of Interaction (Extended Abstract)

Stéphane Gimenez, Georg Moser
University of Innsbruck

February 8, 2016

Abstract

We analyze the complexity of functional programs written in the interaction-net computation model, an asynchronous, parallel and confluent model that generalizes linear-logic proof nets. Employing user-defined *sized* and *scheduled* types, we certify concrete time, space and space–time complexity bounds for both sequential and parallel reductions of interaction-net programs by suitably assigning complexity potentials to typed nodes. The relevance of this approach is illustrated on archetypal programming examples. The provided analysis is precise, compositional and is, in theory, not restricted to particular complexity classes.

1 Introduction

Complexity analysis provides bounds on the amount of resources required for a computation (chiefly time or space) relative to an input size. To ensure compositionality, we analyze properties of outputs, relying on user-defined types which have been enriched with size and, for parallel reductions, timing information.

We base our study on interaction nets, which provide a tangible cost model for both sequential *and* parallel reductions. Interaction nets have been used as an execution platform for functional programs [Mac00; Sou01; Mac04], as a conceptual device for the optimal implementation of the λ-calculus [Lam90; GAL92; AGN96], as a general purpose higher-order language [Fer+09; Gim09], and as a model of distributed computation exemplified by asynchronous abstract hardware [Lip09].

Interaction nets provide a Turing-complete computation model, allow a complexity analysis of sequential reductions of (higher-order) functional programs and allow a reasonably painless extension to parallel reductions, an area typically ignored in the literature (see [HS15] for the exception to the rule). Moreover, as interaction nets incorporate distributed computation, they are of relevance for the study of modern hardware platforms. Hence, a static complexity analysis of interaction nets is also of interest in its own right.

In addition to providing an analysis for space and time complexity separately, we studied *space–time* complexities, i.e., space occupation as a function of time. On the one hand this is a neat technical tool, as certification of precise time or space complexity bounds for sequential and parallel reductions come as very easy corollaries. On the other hand, an accurate prediction of the space–time complexity could prove useful in the application of interaction-net technology in the context of distributed computation.

For sequential reductions, user-defined *sized types* allow to keep track of arbitrarily chosen size measures for intermediate results. From this we obtain a compositional analysis for the space–time complexity of interaction nets which relies on a suitable assignment of potentials to nodes.

To address parallel reductions, we use timing annotations which can be combined with size annotations to control the schedule of the computation. The resulting *scheduled types* express guaranteed or (for inputs) expected time limits on data availability. Based on this we obtain again a precise analysis of space–time complexities for parallel reductions.

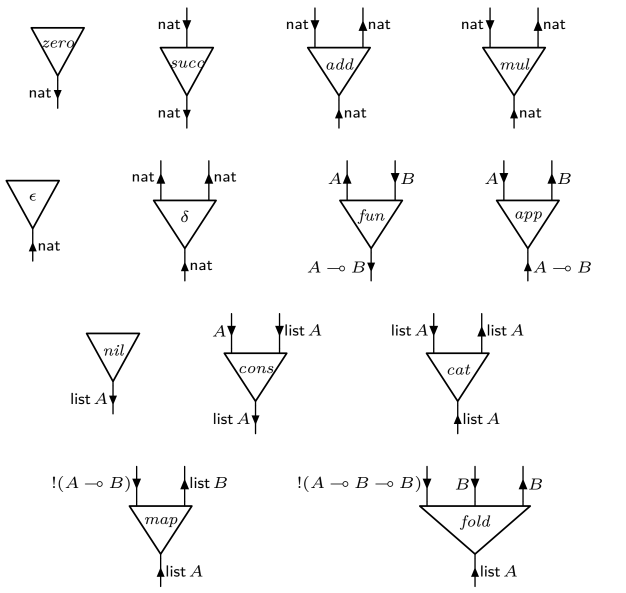
This work has been published in the proceedings of the 43$^{\text{rd}}$ annual symposium on *Principles of Programming Languages* [GM16]. Details and additional content are available in an extended technical report [GM15].

2 Interaction Nets and Their Parallel Reduction

We assume familiarity with interaction nets as described by Lafont in [Laf90]. We first showed that interaction nets (without boxes) form a *reasonable* [Boa90] cost model for time. Computations on Turing machines can be simulated step by step with interaction nets, while, conversely, computations of nets can be performed on a Turing machine in polynomial time. The space required for this representation differs from the number of nodes by the required logarithmic factor needed to encode node identifiers.

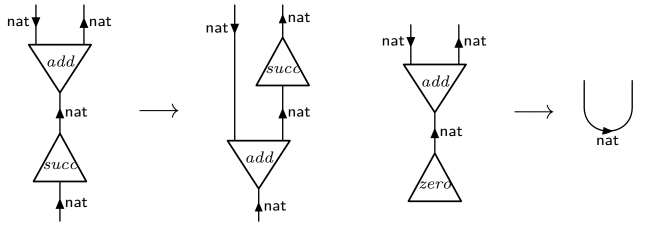
Typed Interaction. The types which we consider are syntactic expressions built from base types, which may expect other types as arguments, and polymorphic variables denoted by A, B, etc., which can be instantiated at will.

As running examples, we consider the following set of typed primitives for natural numbers, abstractions and lists where nat (nullary), $\multimap$ (infix binary), list (unary) and $!$ (unary, called *exponential type* and used to mark polymorphically replicable data) are used as base types. This includes most of the essential ingredients of a typical functional programming language. Yet, our methodology is not restricted to this set of symbols and associated reduction rules.



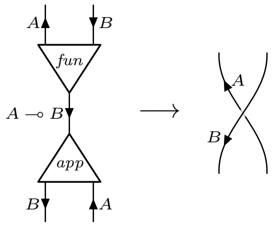
In the above *typing schemes* (one has been provided for every symbol), ports have been oriented and attributed a type. Typically, nodes used as type constructors may admit inputs as auxiliary ports and they output an object of the corresponding type on their principal port. Other nodes can be regarded as functions that seek to interact with their first arguments (provided as inputs on their principal ports), may expect more arguments as additional inputs on auxiliary ports and may output any number of results on remaining auxiliary ports.

Reduction rules for addition of natural numbers are usually defined as follows:



In a *typed interaction-net system*, symbols are provided together with typing schemes and all reduction rules $L \longrightarrow R$ are assumed to be typed and to preserve the types of their interfaces.

Replication. Nodes *fun* and *app* together with the following reduction rule suffice to encode the linear λ-calculus.



In order be used as an expressive higher-order language similar to the full λ-calculus, polymorphic replication is necessary. Various implementations exist in interaction nets; some rely on an infinite family of sharing nodes (as in sharing graphs [GAL92]); other use special devices called *boxes* which, strictly speaking, are not interaction-net nodes. When associated to weak reduction rules, the reduction of boxes admits the diamond property and moreover corresponds quite closely to the weak β-reductions implemented by usual functional programming languages. All the content presented in this paper is compatible with the use of such boxes.

Timed Interaction. We consider a generalization of interaction-net systems to timed reduction rules $L \xrightarrow{d} R$, in which the label $d \geq 0$ denotes a time duration in a chosen time domain T, which can either be discrete or continuous. This allows to define a *timed sequential reduction* $\xrightarrow{t}_s$ as well as a *timed parallel reduction* $\xrightarrow{t}_p$ of duration t in which the firing of a redex begins as soon as it is created.

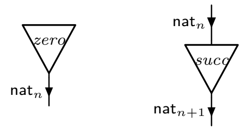
Cost Model. Given a timed reduction $\longrightarrow$ and a notion of space occupation $|\cdot|$ for nets (typically, the number of nodes) in a chosen space domain S, we say that N admits:

- $\tau \in T$ as a time bound if whenever $N \xrightarrow{t} M$, $t \leq \tau$.
- $\sigma \in S$ as a space bound if whenever $N \xrightarrow{t} M$, $|M| \leq \sigma$.
- $\gamma : T \to S$ as a space–time bound if whenever $N \xrightarrow{t} M$, $|M| \leq \gamma(t)$.

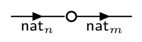
In the following sections, we present methods to compute such bounds for any net in a given interaction-net system, both for the timed sequential reduction $\longrightarrow_s$ and the timed parallel reduction $\longrightarrow_p$. These methods rely on user-defined assignments of potentials to typed nodes, which must be provided together with the defined interaction-net system. We will illustrate the use of these methods by providing such assignments for all the nodes we have introduced.

3 Sized Types and Semantic Complexity

The usual notion of typing provides information concerning the expected shape of inputs and outputs. In order to control the size of natural numbers we introduce a type nat_n for natural numbers whose value is bounded by n, thanks to the following typing schemes:

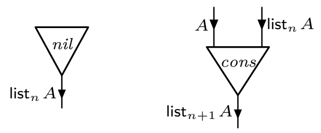


Given that any object of type nat_n can also be given type nat_m if $n \leq m$, we say that nat_n is a subtype of nat_m, written $\mathsf{nat}_n \trianglelefteq \mathsf{nat}_m$. The conversion from nat_n to nat_m for $n \leq m$ can be performed using an explicit type-conversion node:



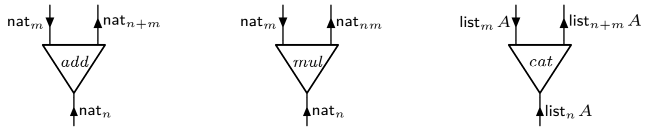
Its reduction rules, are considered instantaneous (i.e. they are attributed time duration 0) because type conversions are not required for actual computation. Subtyping is essential and used to convert a strong type constraint to a weaker constraint.

To control the size of lists we can similarly introduce a type $\mathsf{list}_n A$ for lists whose length is bounded by n, thanks to the following typing schemes:

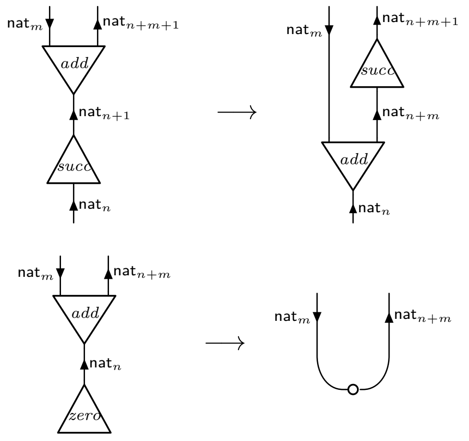


The depth or the number of nodes or various particular size measures of other tree-like data structures could be tracked in the same fashion. More elaborate data structures, e.g., difference lists from [Laf90], can be handled as well.

The following operations on natural numbers and lists can be assigned the following typing schemes:



Addition reductions satisfy typing requirements as follows (we always use the most generic typing for left-hand sides in order to handle all possible valid interactions):



Addition and multiplication can be defined computationally in many different ways. All implementations possess their own particular computational complexity properties (each could be more efficient in a given context), but all additions (respectively, all multiplications) are semantically equivalent and their outputs share the same size bound property, as expressed by their common typing scheme.

4 Sequential Computational Complexity Analysis

We proved a space–time complexity theorem for the sequential reduction, which can be turned into separate space complexity and time complexity theorems.

Corollary 1 (Sequential Time Complexity) *Associate $\tau_c \geq 0$, called time potential, to every typed node c, such that $\sum_{c\in L} \tau_c \geq \sum_{c\in R} \tau_c + d$ for every reduction rule $L \xrightarrow{d} R$.*

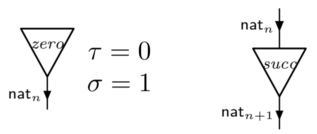
$$N \xrightarrow{t}_s M \implies t \leq \sum_{c\in N} \tau_c$$

Assuming that every node c has been assigned a space-occupation weight $|c| \geq 0$ (weights can be chosen arbitrarily), we define the space occupation of a net as the sum of its nodes' weights: $|N| = \sum_{c\in N} |c|$.

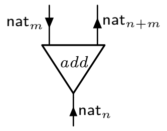
Corollary 2 (Sequential Space Complexity) *Associate $\sigma_c \geq |c|$, called space potential, to every typed node c, such that $\sum_{c\in L} \sigma_c \geq \sum_{c\in R} \sigma_c$ for every reduction rule $L \xrightarrow{d} R$.*

$$N \xrightarrow{t}_s M \implies |M| \leq \sum_{c\in N} \sigma_c$$

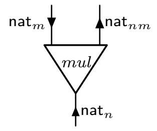
For lack of precise knowledge about the hardware that may host the computation, we chose here to work with unitary reduction durations and to assign a unitary space-occupation weight to all nodes. The following potentials are associated to the displayed typing schemes and parameterized in their size variables. They are compatible with all reduction rules and therefore provide sequential time (τ), space (σ) complexity measures:



$\tau = 0$ $\sigma = 1$; $\tau = 0$ $\sigma = 1$



$$\tau = n + 1 \quad \sigma = 1$$



$$\tau = 2nm + 2n + m + 2 \quad \sigma = nm + n + 1$$

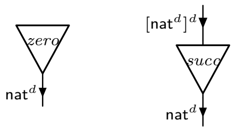
These potentials, which annotate the above natural number library, allow one to deduce sequential resource complexity bounds for programs built with these primitives.

5 Scheduled Types and Productivity

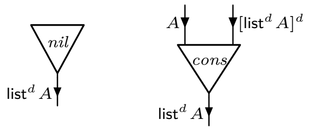
For the care of fully parallel reductions, we distinguish several categories of natural numbers depending on the pace at which they are computed. A natural number admits type nat^d if its first constructor is available now and one additional constructor will be made available after every parallel reduction of duration $d \geq 0$ (or faster).

This corresponds to the following typing scheme definitions, in which we use a bracket notation to denote *delayed types*:

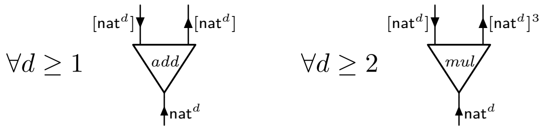
any object of type $[A]^t$ will become an A after a parallel reduction of duration $t \geq 0$ (defaulting to 1 if omitted). We assume syntactic equalities $A = [A]^0$ and $[[A]^{t_1}]^{t_2} = [A]^{t_1+t_2}$.



Similarly we can define a type $\mathsf{list}^d A$ for linearly produced lists with pace d as follows:

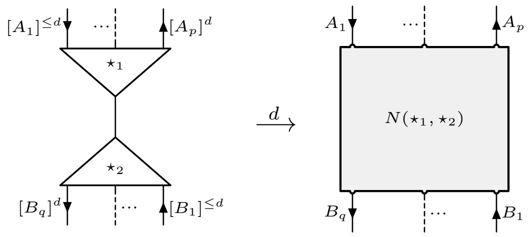


Restricting ourselves to unitary time reductions, we showed that implementations of addition and multiplication exist with the following interfaces:



It is assumed that:

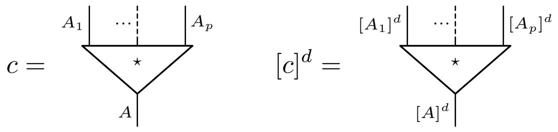
- In typing schemes and left-hand sides of reduction rules, principal ports are assigned undelayed types (i.e. the root is a type variable or a base type).
- The initial delays present in the outputs of a reduction rule are reduced by the rule's duration upon firing. Input delays may but need not be reduced by the full amount of the reduction rule's duration. This scheduling property can be summarized as follows:



In particular, it is always assumed that top-level output delays in the interface of a redex are greater than the assigned duration of the corresponding reduction rule.

Two typing conveniences exist for scheduled types:

- Subtyping: Assuming A is the type associated to a tree-like data structure (i.e. constructors only have inputs, which includes nat and list types, but excludes for example difference lists as in [Laf90] or abstractions), we have $A \trianglelefteq [A]^t$. One can safely use an object of type A where an object of type $[A]^t$ is expected.
- Delayed computation: The execution of a net can be delayed by delaying all the types in its interface (independently of their orientations). For a single node, this allows to instantiate any typing scheme c as $[c]^d$:



Within those conditions, the scheduling property presupposed for reduction rules also holds for the fully parallel reduction of whole nets. If we assume that inputs are available within expected schedules, the parallel reduction of a net will produce outputs in accordance with the schedules that correspond to their types. For example, an output wire typed nat^d will output natural number constructors with pace d.

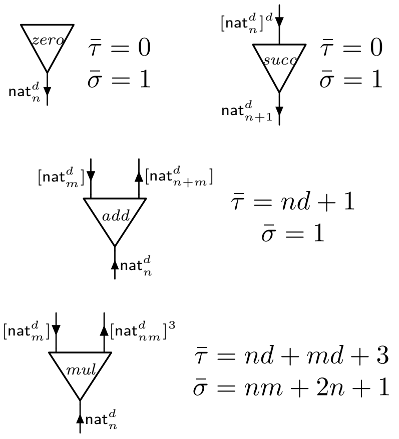
6 Parallel Computational Complexity Analysis

We proved a space–time complexity theorem for the fully parallel reduction, which can then be turned into a separate time complexity theorem. We rely on schedule-typed nodes, i.e. nodes with scheduled typing schemes that include delay connectives and meet the requirements that have been defined in the previous section.

Corollary 3 (Parallel Time Complexity) *Associate $\bar{\tau}_c \geq 0$, called parallel time potential, to every schedule-typed node c, such that $\bar{\tau}_{[c]^d} \geq \bar{\tau}_c + d$ and $\max_{c \in L} \bar{\tau}_c \geq \max_{c \in R} \bar{\tau}_c + d$ for every reduction rule $L \xrightarrow{d} R$.*

$$N \xrightarrow{t}_p M \implies t \leq \max_{c \in N} \bar{\tau}_c$$

With the same assumptions as for the sequential reduction, the following potentials are compatible with reduction rules and therefore provide parallel time ($\bar{\tau}$) and space ($\bar{\sigma}$) complexity measures.

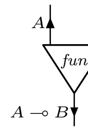


These potentials allow one to deduce parallel resource bounds for programs built with these primitives.

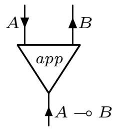
7 Case Study: Higher Order

We can analyze higher-order functional programs in a weak sequential cost model, using the interaction-net framework extended by *functorial promotion boxes* and the associated *weakening*, *contraction*, *dereliction* and *digging* nodes. The sized versions of exponential types are simply expressed as multisets of types. This allows the use of resources to be heterogeneous. We use $!_{i<n}A_i$ as syntactic sugar to denote the multiset $\{A_0, \ldots, A_{n-1}\}$ and write $!_n A$ for homogeneous multisets that contain a single element with multiplicity n. The empty multiset is denoted by $\varnothing$ and the union by $\uplus$. Space occupation of boxes and the duration of box reductions are assumed to be unitary (which is arguably an important simplification, but one similar to attributing a constant cost to a β-reduction).

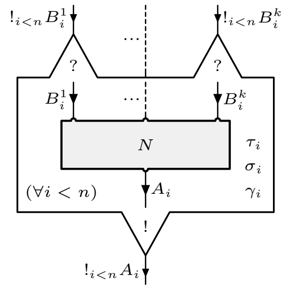
Sized typing schemes and potentials can be assigned as follows. In particular, typing the interface of a box with multisets of size n (they must have the same size) requires typing its contents n times. Requested types for the interface of the contents have been indexed by $i \in [0, n-1]$. Each typing of the contents corresponds recursively to some resource usage τ_i (or σ_i). The potential τ (or σ) of the box is expressed as a function of these resource usages.



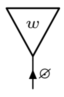
$\tau = 0$
$\sigma = 1$



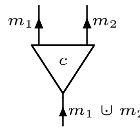
$\tau = 1$
$\sigma = 1$



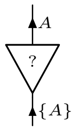
$\tau = \sum_{i<n} \tau_i$
$\sigma = 1 + \sum_{i<n} \sigma_i$



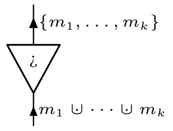
$\tau = 1$
$\sigma = 1$



$\tau = 1$
$\sigma = 1$

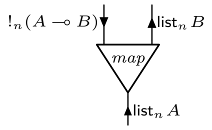


$\tau = 1$
$\sigma = 1$

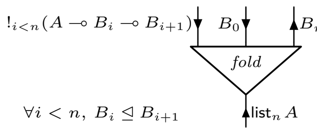


$\tau = 1$
$\sigma = 1$

Multiset inclusion is admissible as subtyping. One can verify that the usual *map* and *fold* admit the following typing schemes and potentials:



$\tau = 2 + 4n$
$\sigma = 1 + 4n$

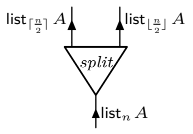


$!_{i<n}(A \multimap B_i \multimap B_{i+1})$
B_0
B_n
fold
$\forall i < n,\ B_i \trianglelefteq B_{i+1}$
$\mathsf{list}_n\, A$

$\tau = 2 + 4n$
$\sigma = 1 + 4n$

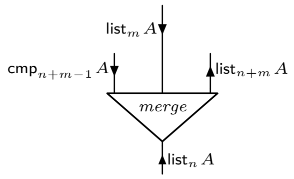
Resource usages of functional arguments to *map* and *fold* are taken into account externally as the potential of the boxes which hold them. Their computation rely on the provided typing schemes.

Merge sort. If $\mathsf{cmp}_n A = !_n(A \multimap A \multimap \mathsf{bool})$ is the type of comparison functions which can be called n times, and $f : \mathbb{N} \to \mathbb{N}$ is recursively defined as $f(n) = n - 1 + f(\lfloor \frac{n}{2} \rfloor) + f(\lceil \frac{n}{2} \rceil)$ (satisfying, $f(n) \leq n \log_2 n$), our methodology allowed us to prove that *merge sort* can be implemented with the following set of nodes and associated resource bounds:

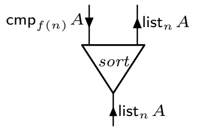


$\mathsf{list}_{\lceil \frac{n}{2} \rceil}\, A$
$\mathsf{list}_{\lfloor \frac{n}{2} \rfloor}\, A$
split
$\mathsf{list}_n\, A$

$\tau = n + 1$
$\sigma = 1$



$\tau = 3n + 3m + 2$
$\sigma = 4$



$\tau = 4f(n) + 10n - 8$
$\sigma = 8n - 7$

8 Conclusion

Because input-focused complexity analysis is not compositional, analyzing properties of outputs is a necessary complement to analyzing time or space requirements. Size annotations can be added to types and validated by a suitable typing of reduction rules. For parallel reductions, schedule-bound transmission of (partial) data composes easily and ensures productivity: timing assumptions on inputs entail timing guaranties on outputs. From there, we straightforwardly obtained sequential and parallel complexity bounds by assigning potentials to typed interaction-net nodes.

In particular, complexity analysis of parallel reductions may be used to optimize the dispatch and scheduling of computation tasks in distributed environments.

To conclude, inspired by bounded linear logic [GSS92], we were able to extend the complexity analysis to interesting higher-order programs. As it turns out, *merge sort* can be typed $!_{n \log_2 n}(A \multimap A \multimap \mathsf{bool}) \multimap \mathsf{list}_n\, A \multimap \mathsf{list}_n\, A$, in which the index on the linear logic modality ensures a concrete $n \log_2 n$ bound on the number of calls to the comparison function. Using a refined version of typing for nets [GM13], we plan to extend our computational complexity results for higher-order programs to non-weak and optimal sequential reductions as well as to parallel reductions.

References

[Boa90] P.v. Emde Boas. "Machine Models and Simulation". In: *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. MIT Press, 1990, pp. 1–66.

[Laf90] Yves Lafont. "Interaction Nets". In: *POPL*. 1990, pp. 95–108. DOI: 10.1145/96709.96718.

[Lam90] John Lamping. "An Algorithm for Optimal Lambda Calculus Reduction". In: *POPL*. 1990, pp. 16–30. DOI: 10.1145/96709.96711.

[GSS92] J-Y. Girard, A. Scedrov, and P. Scott. "Bounded Linear Logic: A Modular Approach to Polynomial-Time Computability". In: *TCS* 97.1 (1992), pp. 1–66.

[GAL92] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. "Linear Logic without Boxes". In: *LICS*. 1992, pp. 223–234. DOI: 10.1109/LICS.1992.185535.

[AGN96] Andrea Asperti, Cecilia Giovannetti, and Andrea Naletto. "The Bologna Optimal Higher-Order Machine". In: *Journal of Functional Programming* 6.6 (1996), pp. 763–810. DOI: 10.1017/S0956796800001994.

[Mac00] Ian Mackie. "Interaction nets for linear logic". In: *Theoretical Computer Science* 247.1-2 (2000), pp. 83–140. DOI: 10.1016/S0304-3975(00)00198-5.

[Sou01] Jorge Sousa Pinto. "Parallel Evaluation of Interaction Nets with MPINE". In: *RTA*. Vol. 2051. Lecture Notes in Computer Science. Springer, 2001, pp. 353–356. ISBN: 3-540-42117-3. DOI: 10.1007/3-540-45127-7_26.

[Mac04] Ian Mackie. "Efficient Lambda-Evaluation with Interaction Nets". In: *RTA*. Vol. 3091. LNCS. 2004, pp. 155–169. DOI: 10.1007/b98160.

[Fer+09] Maribel Fernández et al. "Recursive Functions with Pattern Matching in Interaction Nets". In: *ENTCS* 253.4 (2009), pp. 55–71. DOI: 10.1016/j.entcs.2009.10.017.

[Gim09] Stéphane Gimenez. "Programmer, calculer et raisonner avec les réseaux de la Logique Linéaire". PhD thesis. 2009. URL: http://pps.jussieu.fr/~gimenez/these.html.

[Lip09] Sylvain Lippi. "Universal Hard Interaction for Clockless Computation. Dem Glücklichen schlägt keine Stunde!" In: *Fundamenta Informaticae* 91.2 (2009), pp. 357–394. DOI: 10.3233/FI-2009-0048.

[GM13] Stéphane Gimenez and Georg Moser. "The Structure of Interaction". In: *CSL*. Vol. 23. 2013, pp. 316–331. ISBN: 978-3-939897-60-6. DOI: 10.4230/LIPIcs.CSL.2013.316.

[GM15] S. Gimenez and G. Moser. *The Complexity of Interaction (Long Version)*. 2015. URL: http://arxiv.org/abs/1511.01838.

[HS15] Jan Hoffmann and Zhong Shao. "Automatic Static Cost Analysis for Parallel Programs". In: *ESOP*. Vol. 9032. LNCS. 2015, pp. 132–157. DOI: 10.1007/978-3-662-46669-8_6.

[GM16] Stéphane Gimenez and Georg Moser. "The Complexity of Interaction". In: POPL 2016. ACM, 2016, pp. 243–255. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837646.