The Complexity of Interaction

Stéphane Gimenez Georg Moser

Institute of Computer Science, University of Innsbruck, Austria

{stephane.gimenez,georg.moser}@uibk.ac.at

Abstract

In this paper, 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.

Categories and Subject Descriptors F.3.2 [*Semantics of programming languages*]: Program Analysis

Keywords Program Analysis, Interaction Nets, Linear Logic, Sequential and Parallel Reductions, Time and Space Bounds.

1. Introduction

Complexity analysis provides bounds on the amount of resources required for a computation (chiefly time or space) relative to an input size.

Suppose that a function call $f(x)$ executes in time $\mathrm{O}(|x|)$ and $g(x)$ executes in time $\mathrm{O}(2^{|x|})$. What is the time complexity of the composition $g \circ f$ of these two functions? Is it perhaps $\mathrm{O}(2^{|x|})$? The real answer is: we don't know. It depends on the size of $f(x)$, which could be logarithmic, linear, or even exponential (for example in a parallel computation model), among other possibilities, with respect to the size of x. The complexity of the composition $g \circ f$ varies consequently:

$$\square\square\square\square \xrightarrow{log} \square\square \xrightarrow{g} a \qquad \mathrm{O}(|x|)$$

$$\square\square\square\square \xrightarrow{id} \square\square\square\square \xrightarrow{g} b \qquad \mathrm{O}(2^{|x|})$$

$$\square\square\square\square \xrightarrow{exp} \square\square\square\square\square\square\square\square\square\square\square\square\square\square\square\square \xrightarrow{g} c \qquad \mathrm{O}(2^{2^{|x|}})$$

Complexity analysis in the traditional sense is therefore not compositional. To ensure compositionality, in this paper, 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 [25, 26, 30], as a conceptual device for the optimal implementation of the λ-calculus [1, 14, 22], as a general purpose higher-order language [7, 9], and as a model of distributed computation exemplified by asynchronous abstract hardware [23].

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 [17] 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 emphasize the study of *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 (Theorem 1). A practical difficulty is the combination of these potentials, because the juxtaposition of two terms that are tied to sequential space–time complexities f and g is a complex convolution $h(t) = \max_{u+v=t} f(u) + g(v)$ where u and v denote the respective times allocated to the reduction of each term. We draw attention to the fact that sequential computation in interaction nets slightly deviates from the standard notion. Traditionally in a sequential deterministic computation, the instruction to be performed next is predefined for the single execution thread. However, interaction-net systems rely on the diamond property to guarantee confluence and thus allow "don't-care non-determinism", which our complexity analysis can handle.

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 (Theorem 2). While typing schemes become more involved in the context of parallel reductions, the space–time complexity of a juxtaposition is in this case easily expressed as a sum $h(t) = f(t) + g(t)$.

The remainder of this paper is structured as follows. Section 2 briefly surveys interaction nets and outlines our motivating examples. In Section 3 we define sized types with a semantic complexity analysis in mind. These are employed in Section 4, in which we state and prove our first main theorem concerning computational complexities of sequential reductions. Section 5 introduces sched-

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837646

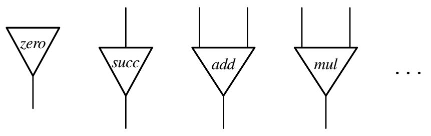
uled types. We use them in Section 6 to establish our second main theorem concerning computational complexities of parallel reductions. In Section 7 we show how our results can be used to analyze weak sequential reductions of higher-order programs. In Section 8 we discuss related work. Finally we conclude in Section 9, where we also report on future work.

Further details and additional content are available in an extended technical report [10].

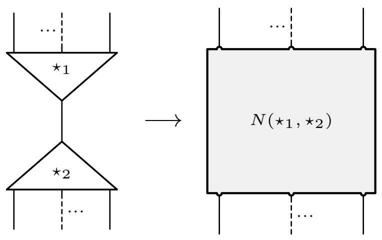
2. Interaction Nets and Their Parallel Reduction

The reader is assumed to be familiar with interaction nets as described by Lafont in [18]. Various formal definitions of interaction nets can be found elsewhere in the literature [27, 29, 33], but an intuitive understanding of the underlying graph-rewriting technology should be sufficient in the context of this paper.

An *interaction-net system* is a pair $(\mathcal{S}, \mathcal{R})$ consisting of a set of *symbols* $\mathcal{S}$ used to label *nodes* and an associated set of *reduction rules* $\mathcal{R}$. Each symbol has an integer *arity* that dictates the number of *auxiliary ports* which nodes labeled with this symbol possess. Additionally, each node possesses exactly one distinguished *principal port*. Graphically, in the following set of labeled nodes, which will be used to represent constructors (*zero*, nullary, *succ*, unary) and operations (*add* and *mul*, binary) over natural numbers, auxiliary ports are located at the top and principal ports are located at the bottom.



An *interaction net* is a graph built from such nodes, where each port can be connected to one other port by one wire. Unconnected ports of a net are called free ports and are collectively referred to as the *interface* of this net. By design, interaction-net redexes consist of two nodes only (whose labels are represented abstractly by $\star_1$ and $\star_2$ in the following picture) that are connected through their principal ports. Reduction rules reduce them in context to a given net:



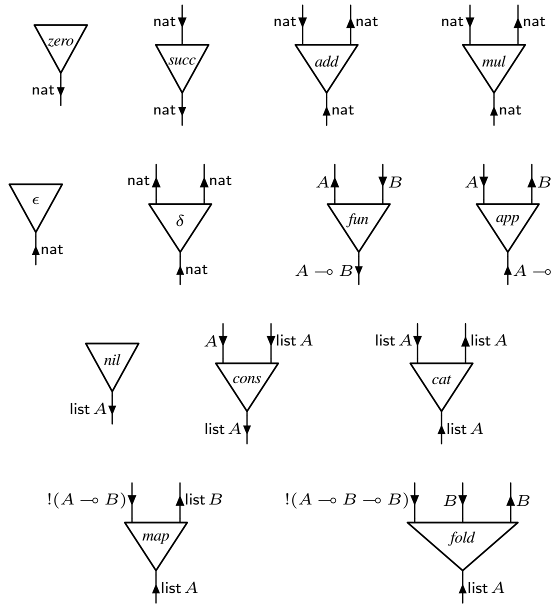
To ensure determinism when firing a single redex, only one reduction rule is allowed per symbol pair and the net $N(\star_1, \star_2)$ is assumed symmetric if $\star_1 = \star_2$. Because redexes cannot overlap, the reduction enjoys the diamond property and can be parallelized easily; normal forms are unique. Examples can be found in [18].

Interaction nets form a simple and realistic computation model as the locating, recording and firing of one redex can all be implemented in constant time and constant space on standard computer architectures.

With respect to sequential reduction, we show [10] that interaction nets (without boxes) form a *reasonable* [5] 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 latter result follows from a straightforward encoding of the net as an adjacency list of nodes. The space required for this representation differs from the number of nodes by the required logarithmic factor needed to encode node identifiers.

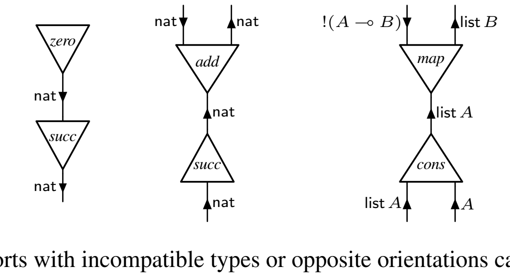
Typed Interaction Types are syntactic expressions built from base types, which may expect other types as arguments, and polymorphic variables denoted by A, B, etc., sometimes parameterized by indices, which can be instantiated at will.

As a running example, we will 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.

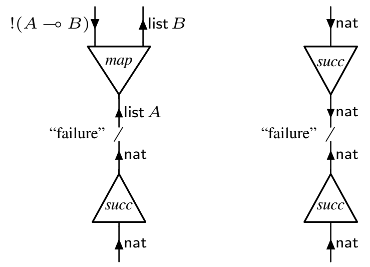


In the above *typing schemes* (one has been provided for every symbol), ports have been oriented and attributed a type. Typically, nodes used as type constructors may admit inputs as auxiliary ports and they output an object of the corresponding type on their principal port. Other nodes can be regarded 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.

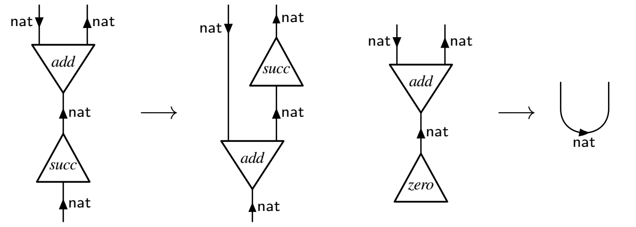
In a *typed interaction net*, wires and free ports are assigned types which must be instances of the types from the typing schemes associated to the symbols that label the nodes to which they are attached. In other words, instances of typing schemes which have been assigned to nodes have to unify on every wire, with matching type orientations:



Ports with incompatible types or opposite orientations cannot be wired together:

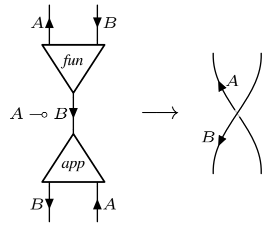


Reduction rules for addition of natural numbers are usually defined as follows:

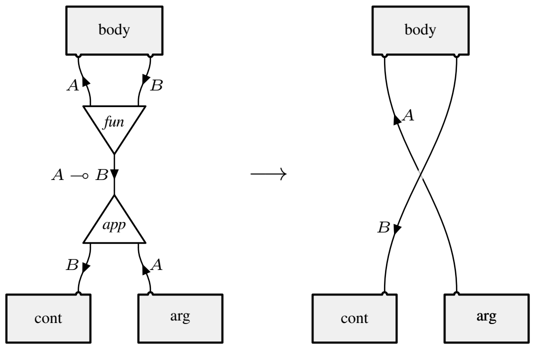


In a *typed interaction-net system*, symbols are provided together with typing schemes and all reduction rules $L \longrightarrow R$ are assumed to be typed and to preserve the types of their interfaces. This means that L and R are typed interaction nets and the typing of the interface of L matches the one of R. This assumption entails a subject reduction property: any reduct of a typed net can be typed in a way that preserves the typing of its interface.

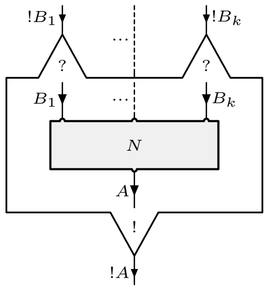
Replication Nodes *fun* and *app* together with the following reduction rule suffice to encode the linear λ-calculus; see [9].

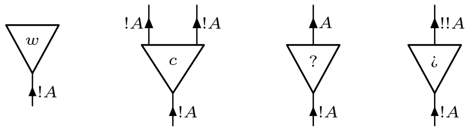


In order to ease the understanding of this rule, virtual body, argument and continuation passed to the function can be conceptualized as a context:

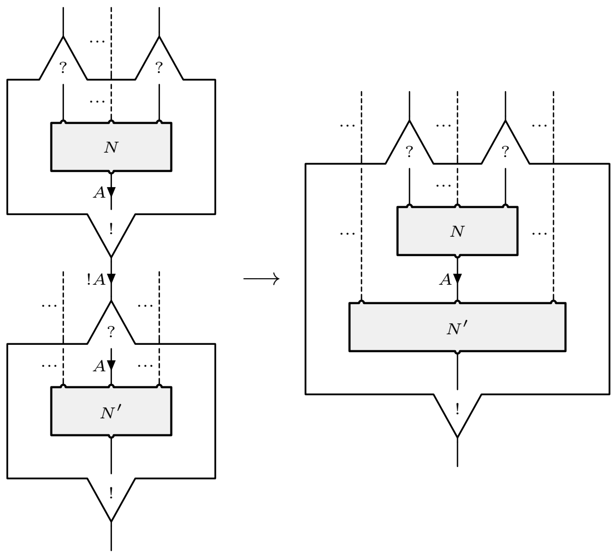


In order be used as an expressive higher-order language similar to the full λ-calculus, polymorphic replication is necessary. Various implementations exist in interaction nets; some rely on an infinite family of sharing nodes (as in sharing graphs [14]); other use special devices called *boxes* which, 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. Namely, we chose to work with *functorial promotion boxes* [24, 28], which are parameterized by a net (represented below as N), together with the usual *weakening* (w), *contraction* (c), *dereliction* ($?$) and *digging* ($¿$) nodes from linear logic.



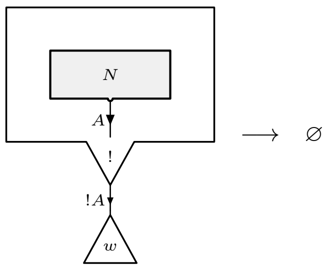


In a weak setting, nets that parameterize boxes are not reduced internally. Boxes are reduced externally and non-closed boxes ($k > 0$) may only merge with other boxes until they are eventually closed (some types are omitted for clarity):

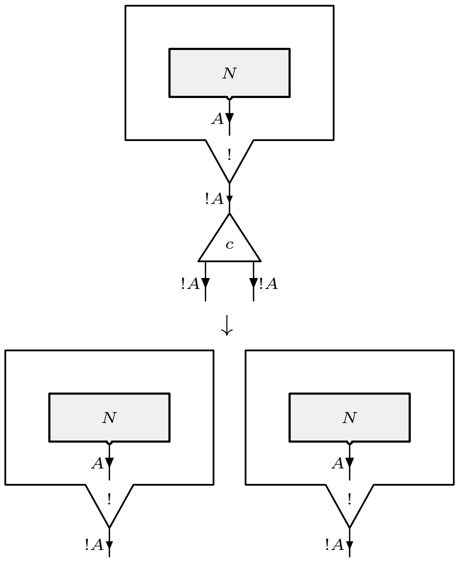


Closed boxes ($k = 0$) can be:

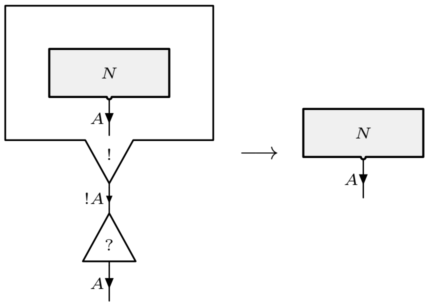
- erased by *weakenings*:



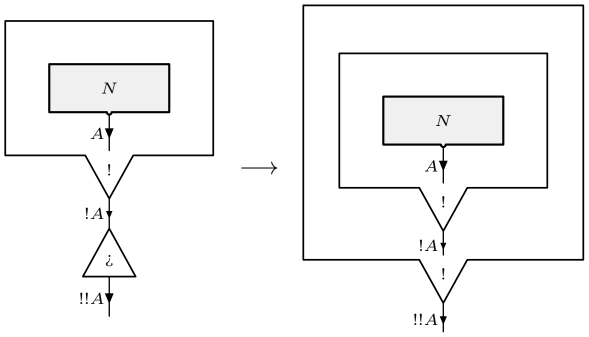
- duplicated by *contractions*:



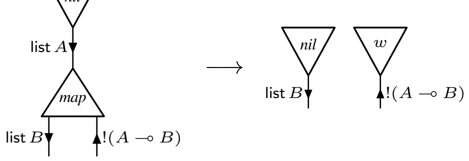
- opened by *derelictions*:

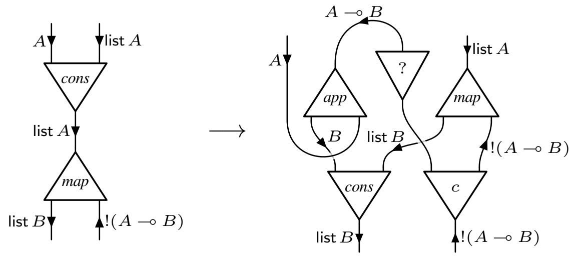


- and doubled by *diggings*:



As an illustration, Figure 1 depicts the 5-step fully parallel reduction (i.e. all available redexes are fired together at once) that reduces to normal form the mapping of the successor constructor to the list of natural numbers $[0, 0]$, according to the following reduction rules:





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.

We define *timed sequential reduction* $\longrightarrow_s$ as the extension of the timed reduction rules generated by the following transitivity property: $N \xrightarrow{t}_s M$ if there exists t_1, t_2 and a net P such that $t = t_1 + t_2$ and $N \xrightarrow{t_1}_s P$ and $P \xrightarrow{t_2}_s M$. The timed sequential reduction satisfies a timed diamond property: if $N \xrightarrow{t_1}_s P_1$ and $N \xrightarrow{t_2}_s P_2$ then there exist M such that $P_1 \xrightarrow{t_2}_s M$ and $P_2 \xrightarrow{t_1}_s M$. This ensures that all reductions to normal form have the same duration.

The definition of *timed parallel reduction* $\longrightarrow_p$ requires labeling every instance of a redex r with a counter value $c(r)$ (defaulting to 0 initially) that will always be less than the time $d(r)$ associated to the corresponding reduction rule. We perform the parallel reduction $N \xrightarrow{t}_p M$ of duration t of a net N (assumed not to be in normal form) by increasing all redex counters by t if all counters satisfy $c(r) + t < d(r)$. Otherwise, by firing simultaneously all redexes which minimize $d(r) - c(r)$, then increasing other counters by the obtained minimal value t_0, and pursuing recursively the reduction for a duration of $t - t_0$. The parallel reduction $\xrightarrow{t}_p$ is strictly deterministic and satisfies the same transitivity property as the sequential reduction.

In this paper, for simplicity, we will only consider examples where reduction rules (with the exception of type-conversion rules which will be defined later) are arbitrarily assigned unitary time durations, and correspond to standard interaction-net systems. The theory however allows one to take into account various implementation details since it is general and applicable to arbitrary durations as well.

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 provide 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

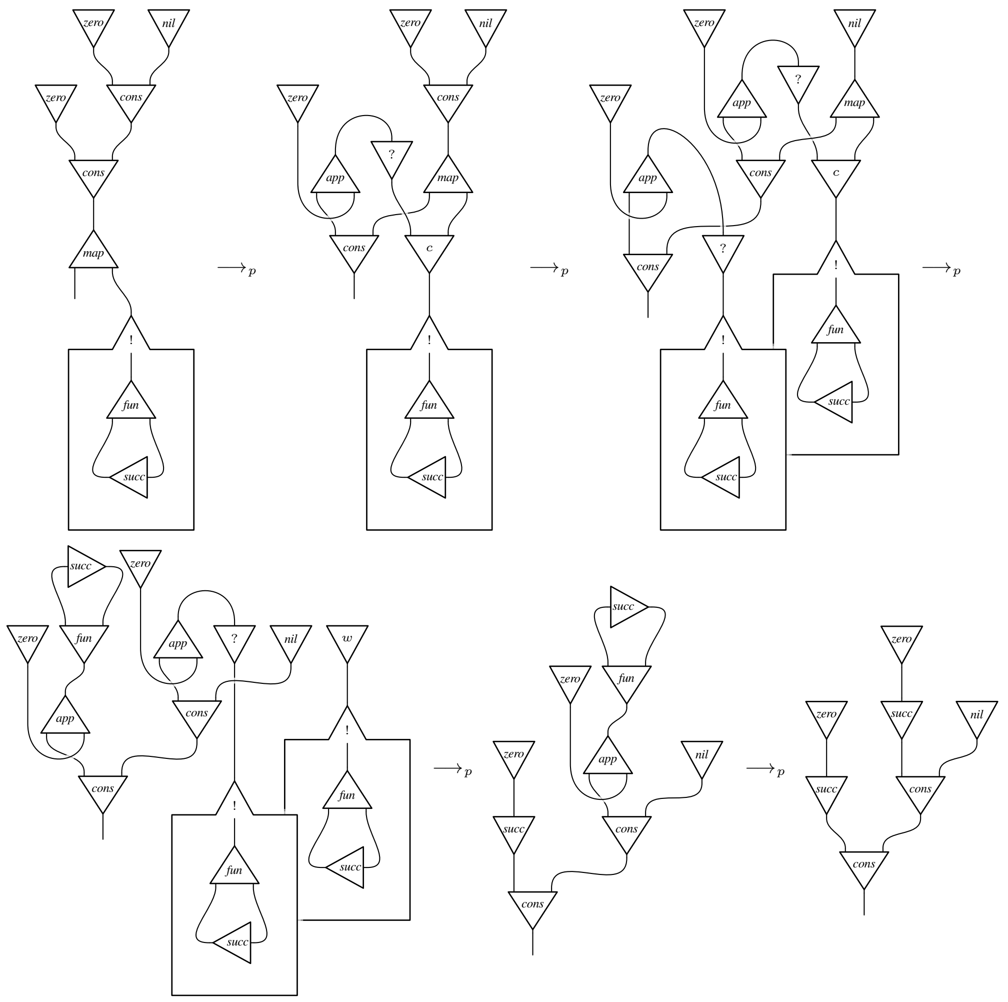
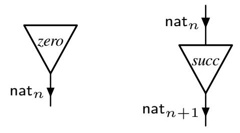
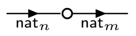


Figure 1: Fully parallel reduction of a functional program

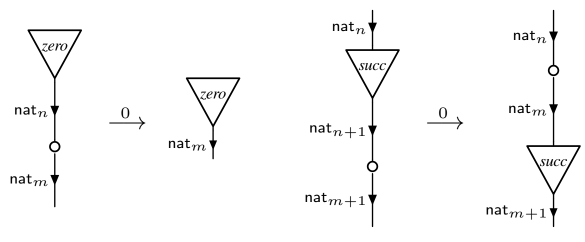
whose value is bounded by n, thanks to the following typing schemes:



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:

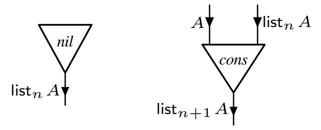


Its reduction rules,



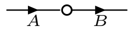
are considered instantaneous (i.e. they are attributed time duration 0) because type conversions are not required for actual computation.

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:

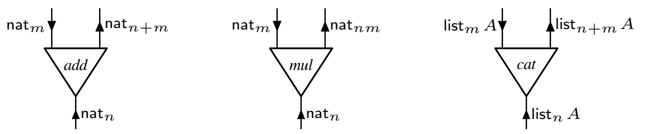


We easily obtain $\mathsf{list}_n A \trianglelefteq \mathsf{list}_m A$ if $n \leq m$ using similar conversion rules. More generally, $\mathsf{list}_n A \trianglelefteq \mathsf{list}_m B$ if $n \leq m$ and $A \trianglelefteq B$. The depth or the number of nodes or various particular size measures of other tree-like data structures could be tracked in the same fashion. More elaborate data structures, e.g., difference lists from [18], can be handled as well.

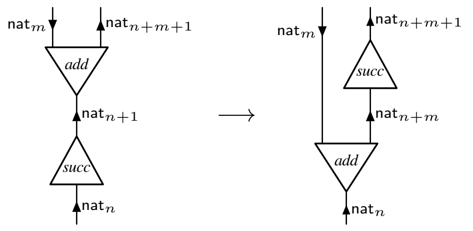
Subtyping is essential and used to convert a strong type constraint to a weaker constraint. Whenever $A \trianglelefteq B$, one is allowed to use an object of type A where an object of type B is expected, using type-conversion nodes of the following shape, which we allow to appear in reduction rules:

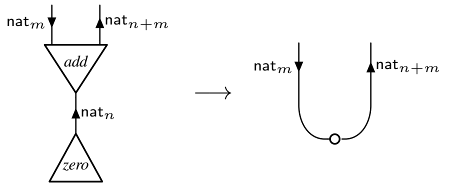


As shown below, the following operations on natural numbers and lists can be assigned the following typing schemes:

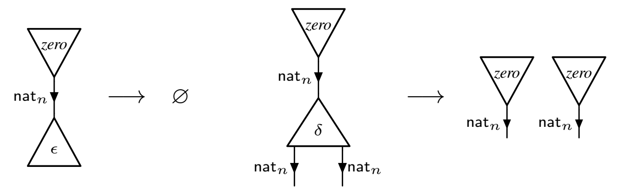


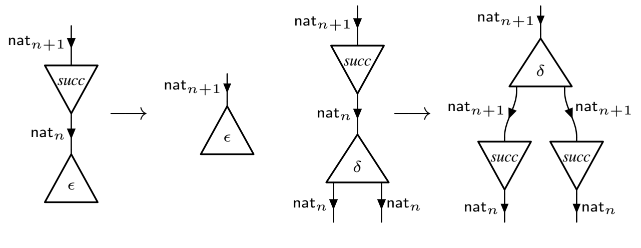
Addition reductions satisfy typing requirements as follows (we always use the most generic typing for left-hand sides in order to handle all possible valid interactions):



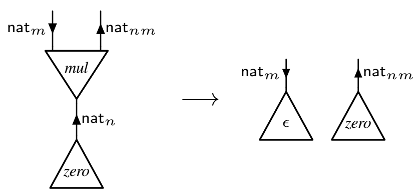


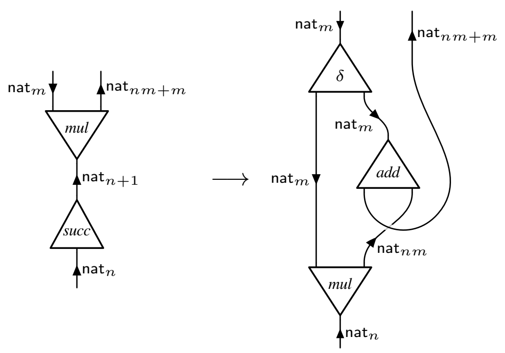
Specialized versions of the usual interaction-net combinators δ and ϵ (see [19], they are also called "Dupl" and "Erase" in [18]) are used to copy or delete natural numbers. Their reductions are the following:



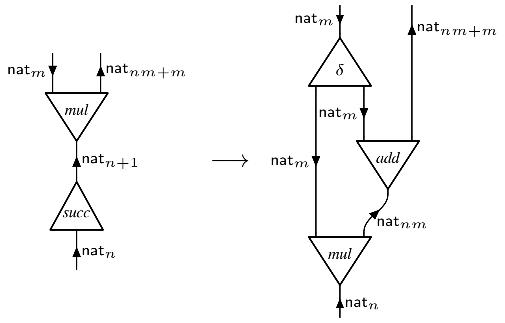


The most standard multiplication (variant 1) is defined below with interaction nets. The adequate typing of its reduction rules validates the size complexity property expressed in its typing scheme.



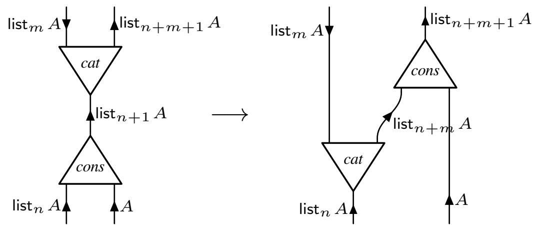


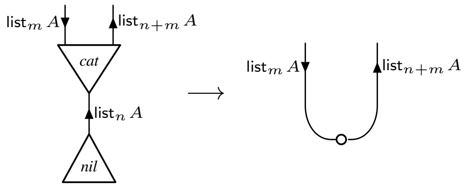
Multiplication (variant 2) is the one presented in [18]. It has different computational properties (it is generally less efficient), but it can be typed similarly. The previous reduction rule is replaced by the following in which the arguments to the recursive call have been swapped:



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.

Last, size bounds for list concatenation are obtained as follows:

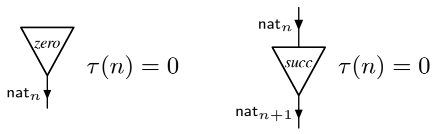


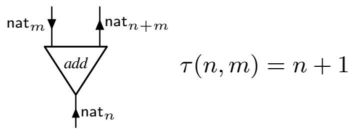


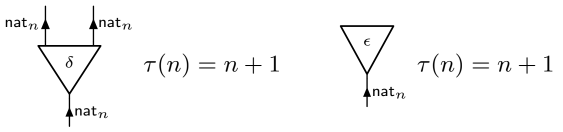
4. Sequential Computational Complexity Analysis

We will prove a space–time complexity theorem for the sequential reduction, which will then be turned into separate space complexity and time complexity theorems. However, we provide first a simple example explaining how our analysis allows one to infer the time complexity of multiplication (variant 2) from those of the operations used in its definition.

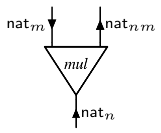
Example 1 *Assuming knowledge of time potentials τ (which, as will be explained soon, correspond closely to time complexities) for the following typed nodes,*







our results ensure that multiplication (variant 2) as defined previously can be attributed potential (and complexity):

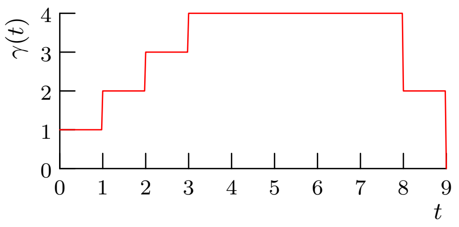


$$\tau_{mul}(n, m) = \frac{n^2 + n}{2} m + 3n + m + 2$$

The only requirement is to check that, in all reduction rules, the sum of the time potentials assigned to typed nodes in the left-hand side is strictly greater than the sum of the time potentials assigned to typed nodes in the right-hand side. In this example, the constraints associated to multiplication reduction rules, $\tau_{mul}(n, m) + 0 > (m + 1) + 0$ and $\tau_{mul}(n + 1, m) + 0 > (nm + 1) + (m + 1) + \tau_{mul}(n, m)$, can be checked or solved easily to obtain the result.

We introduce notations which we will use later to represent concrete space–time complexities. Given a function $f : T \to S$, where T denotes a time domain and S a space domain, we define delayed functions $[f]^d$ as $[f]^d(t) = f(t - d)$, or $[f]^d(t) = 0$ when $t < d$. Iterated sequences are defined recursively as $f \sqsupset_0^d g = f$ and $f \sqsupset_{n+1}^d g = (f \sqsupset_n^d g) + [g]^{(n+1)d}$. Intuitively $f \sqsupset_n^d g$ represents the superimposition of n copies of g, successively delayed by d, to f. Finally, compound iterated sequences recursively extend this definition with successive superimpositions as $f \sqsupset_n^d g \sqsupset_{n_1}^{d_1} g_1 \sqsupset_{n_2}^{d_2} \cdots \sqsupset_{n_p}^{d_p} g_p = (f \sqsupset_n^d g) \sqsupset_{n_1}^{d_1} [g_1]^{nd} \sqsupset_{n_2}^{d_2} \cdots \sqsupset_{n_p}^{d_p} [g_p]^{nd}$. The exponent or index of any $\sqsupset$-symbol defaults to 1 if omitted. We abusively use numerals to denote constant functions.

For example, here is a graphical representation of $\gamma = 1 \sqsupset_3 1 \sqsupset_4 0 \sqsupset_2 -2$. It initially has value 1 and successively undergoes (after unitary delays) 3 increments of 1, then remains constant 4 times, and undergoes 2 decrements of 2.



We define the *sequential convolution* of functions f and g as follows:

$$(f * g)(t) = \max_{u+v=t} f(u) + g(v)$$

This operation is commutative and associative. The generalized sequential convolution of a family of functions $\{ f_i \}_{i \in I}$ can be expressed as:

$$(\coprod_{i \in I} f_i)(t) = \max_{\sum_{i \in I} t_i = t} \sum_{i \in I} f_i(t_i)$$

We will range over all typed nodes of a net N by indexing an operation (e.g. a sum or a sequential convolution) with $c \in N$. Nodes occurring multiple times in N are counted over multiple times. In particular, 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: $|N| = \sum_{c \in N} |c|$.

Theorem 1 (Sequential Space–Time Complexity) *Associate a function $\gamma_c : T \to S$, called space–time potential, to every typed node c, such that $\gamma_c(0) \geq |c|$ and $(\coprod_{c \in L} \gamma_c)(t + d) \geq (\coprod_{c \in R} \gamma_c)(t)$ for every reduction rule $L \xrightarrow{d} R$.*

$$N \xrightarrow{t}_s M \implies |M| \leq (\coprod_{c \in N} \gamma_c)(t)$$

Proof. The potential-decrease property assumed for reduction rules entails the same property for individual sequential reduction steps of a net: $\gamma_L(t + d) \geq \gamma_R(t) \Rightarrow (\gamma * \gamma_L)(t + d) = \max_{u+v=t+d}(\gamma(u) + \gamma_L(v)) \geq \max_{u+v=t}(\gamma(u) + \gamma_L(v + d)) \geq \max_{u+v=t}(\gamma(u) + \gamma_R(v)) = (\gamma * \gamma_R)(t)$. By combining these steps, we obtain $(\coprod_{c \in N} \gamma_c)(t) \geq \cdots \geq (\coprod_{c \in M} \gamma_c)(0) = \sum_{c \in M} \gamma_c(0) \geq |M|$. □

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 $L \xrightarrow{d} R$.*

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

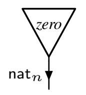
Proof. Choose $\gamma_c(t) = \tau_c - t$ and null space-occupation weights; remark that $(\coprod_{c \in N} \gamma_c)(t) = \sum_{c \in N} \tau_c - t$. □

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 $L \xrightarrow{d} R$.*

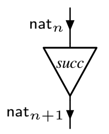
$$N \xrightarrow{t}_s M \implies |M| \leq \sum_{c \in N} \sigma_c$$

Proof. Choose constant functions $\gamma_c(t) = \sigma_c$; remark that $(\coprod_{c \in N} \gamma_c)(t) = \sum_{c \in N} \sigma_c$. □

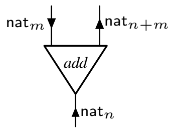
Subtyping conversion rules such as those we have defined satisfy the sequential potential-decrease property; because they have been assigned durations 0, it is sufficient to ensure that the potential assigned to constructors is monotonic with respect to subtyping. For lack of precise knowledge about the hardware that may host the computation, we choose to assign a unitary space-occupation weight to all nodes. For our present needs, constructors will be assigned constant witness potentials, but they could be assigned different potentials if we were to perform amortized cost analysis in full. 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) and space–time (γ) complexity measures:



$$\tau = 0 \quad \sigma = 1 \quad \gamma = 1$$

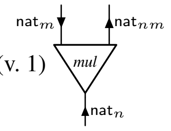


$$\tau = 0 \quad \sigma = 1 \quad \gamma = 1$$

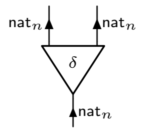


$$\tau = n + 1 \quad \sigma = 1 \quad \gamma = 1 \sqsupset_n 0 \sqsupset -2$$

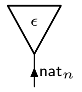
(v. 1)



$$\tau = 2nm + 2n + m + 2 \quad \sigma = nm + n + 1 \quad \gamma = 1 \sqsupset_{nm+n} 1 \sqsupset_{nm+1} 0 \sqsupset_m -1 \sqsupset_{n+1} -2$$



$$\tau = n + 1 \quad \sigma = n + 1 \quad \gamma = 1 \sqsupset_n 1 \sqsupset 0$$



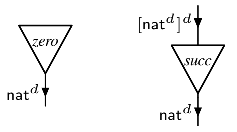
$$\tau = n + 1 \quad \sigma = 1 \quad \gamma = 1 \sqsupset_n -1 \sqsupset -2$$

Example of a concrete application to program complexity certification If we consider the net built by providing interaction-net representations of two natural numbers bounded respectively by n and m as arguments to a *mul* node, the combined time and space potentials of all the nodes are bounded respectively by $\tau = 2nm+2n+m+2$ and $\sigma = nm + 2n + m + 3$ (constructors in the representation of natural number i possess themselves a combined potential of $\tau = 0$ and $\sigma = i + 1$). According to Corollaries 1 and 2, these combined potentials constitute concrete bounds on the sequential time and space needed to perform the multiplication as a function of the size of its arguments. Furthermore, the bounds obtained through this method are precise.

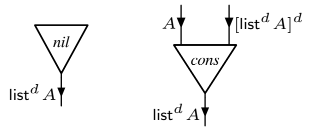
5. Scheduled Types and Productivity

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 $[[A]^{t_1}]^{t_2} = [A]^{t_1+t_2}$.

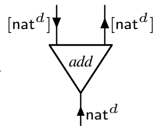


Similarly we can define a type list$^d\,A$ for linearly produced lists with pace d as follows:

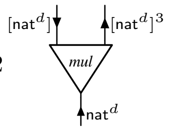


Restricting ourselves to unitary time reductions, we will show that implementations of addition, multiplication and list concatenation exist with the following interfaces:

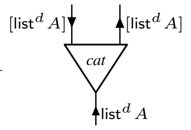
$\forall d \geq 1$



$\forall d \geq 2$

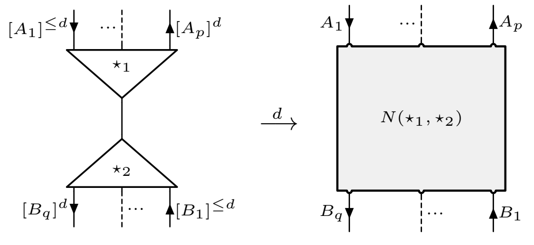


$\forall d \geq 1$



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:

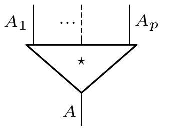


In particular, it is always assumed that top-level output delays in the interface of a redex are greater than the assigned duration of the corresponding reduction rule.

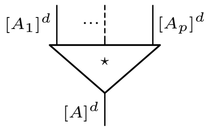
Two typing conveniences exist for scheduled types:

- 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 [18] 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. In particular, we deduce nat$^{d_1} \trianglelefteq$ nat^{d_2} if $d_1 \leq d_2$, and list$^{d_1}\,A \trianglelefteq$ list$^{d_2}\,B$ if $d_1 \leq d_2$ and $A \trianglelefteq B$.
- 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 $[c]^d$:

$c =$

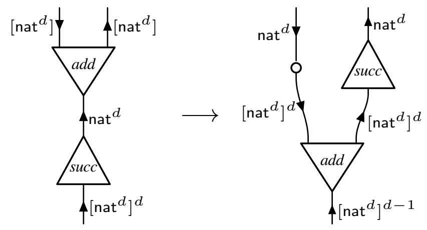


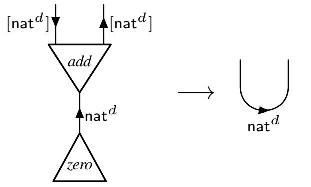
$[c]^d =$



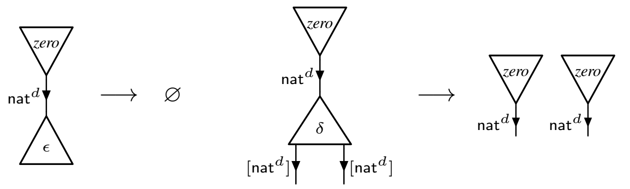
Within those conditions, the scheduling property presupposed for reduction rules also holds for the fully parallel reduction of whole nets. If we assume that inputs are available within expected schedules, the parallel reduction of a net will produce outputs in accordance with the schedules that correspond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

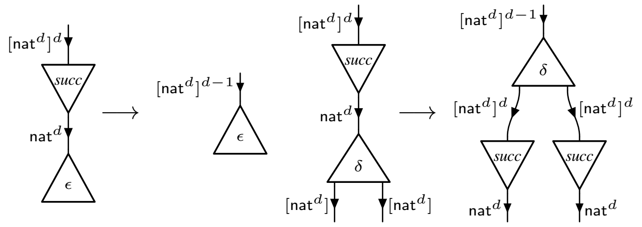
Reduction rules for addition of natural numbers can be typed as follows ($d \geq 1$):



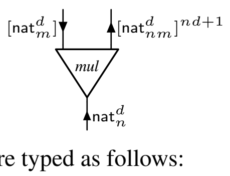


The typing and reduction rules of the specialized versions of δ and ϵ to natural numbers are straightforward ($d \geq 1$):

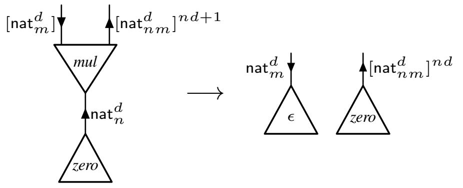


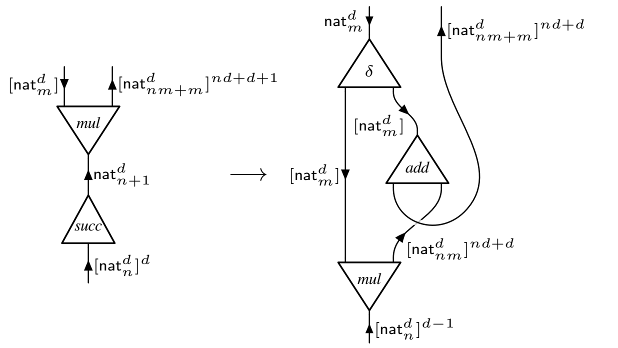


For multiplication (variant 1) of natural numbers, the timing analysis has to rely on the size analysis. The output is produced after a significant delay that depends (linearly) on the size of the first parameter. This example illustrates the combined use of size and timing annotations which is generally required to obtain precise complexity bounds. For $d \geq 1$, we obtain:

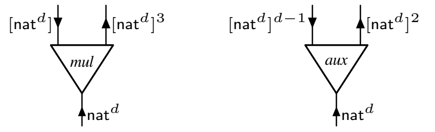


Reduction rules are typed as follows:

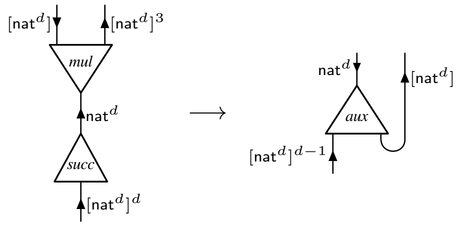


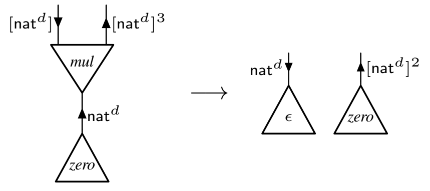


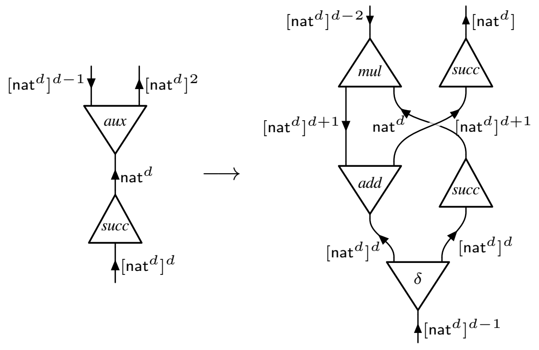
Multiplication (variant 3) of natural numbers is defined using an auxiliary node and constitutes in most cases an interesting improvement in a parallel model as it offers a constant delay. For $d \geq 2$, we obtain:

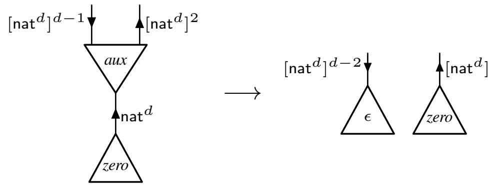


Reduction rules are the following:

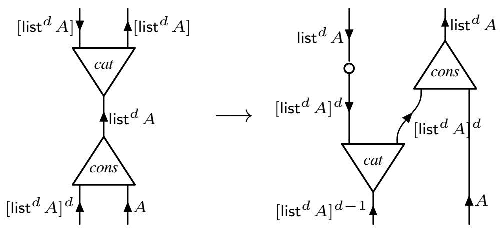


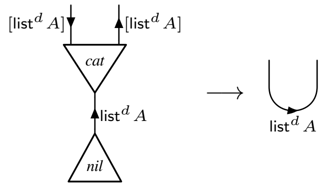






Last, reduction rules for list concatenation are typed as follows:





6. Parallel Computational Complexity Analysis

We prove a space–time complexity theorem for the fully parallel reduction, which is then 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.

Theorem 2 (Parallel Space–Time Complexity) *Associate a function $\bar{\gamma}_c : T \to S$, called parallel space–time potential, to every schedule-typed node c, such that $\bar{\gamma}_c(0) \geq |c|$, $\bar{\gamma}_{[c]^d}(t+d) \geq \bar{\gamma}_c(t)$ and $(\sum_{c \in L} \bar{\gamma}_c)(t+d) \geq (\sum_{c\in R} \bar{\gamma}_c)(t)$ for every reduction rule $L \xrightarrow{d} R$.*

$$N \xrightarrow{t}_p M \implies |M| \leq (\sum_{c \in N} \bar{\gamma}_c)(t)$$

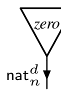
Proof. The potential-decrease property assumed for reduction rules entails the same property for atomic parallel reduction steps of a net. Hence, by combining these steps, we obtain $(\sum_{c\in N} \bar{\gamma}_c)(t) \geq \cdots \geq (\sum_{c \in M} \bar{\gamma}_c)(0) = \sum_{c\in M} \bar{\gamma}_c(0) \geq |M|$. □

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 $L \xrightarrow{d} R$.*

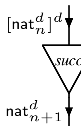
$$N \xrightarrow{t}_p M \implies t \leq \max_{c \in N} \bar{\tau}_c$$

Proof. Choose $\bar{\gamma}_c(t) = \bar{\tau}_c - t$ and null space-occupation weights; remark that $(\sum_{c\in N} \bar{\gamma}_c)(t) \geq 0$ implies $t \leq \max_{c \in N} \bar{\tau}_c$. □

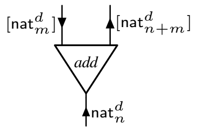
With the same assumptions as for the sequential reduction, the following potentials are compatible with reduction rules and therefore provide parallel time ($\bar{\tau}$) and space–time ($\bar{\gamma}$) complexity measures. Notice that this analysis reveals that multiplication (variant 3), which has an output of quadratic size, can be performed in linear time with a parallel reduction.



$\bar{\tau} = 0$
$\bar{\sigma} = 1$
$\bar{\gamma} = 1$

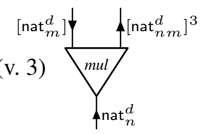


$\bar{\tau} = 0$
$\bar{\sigma} = 1$
$\bar{\gamma} = 1$

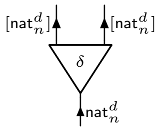


$\bar{\tau} = nd + 1$
$\bar{\sigma} = 1$
$\bar{\gamma} = 1 \sqsupset^d_n 0 \sqsupset -2$

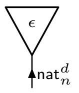
(v. 3)



$\bar{\tau} = nd + md + 3$
$\bar{\sigma} = nm + 2n + 1$
$\bar{\gamma} = 1 \sqsupset^d_n (-1 \sqsupset 3 \sqsupset^d_m 1 \sqsupset 0 \sqsupset -2) \sqsupset^d_m -1 \sqsupset -2$



$\bar{\tau} = nd + 1$
$\bar{\sigma} = n + 1$
$\bar{\gamma} = 1 \sqsupset^d_n 1 \sqsupset 0$



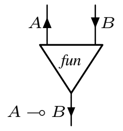
$\bar{\tau} = nd + 1$
$\bar{\sigma} = 1$
$\bar{\gamma} = 1 \sqsupset^d_n -1 \sqsupset -2$

For parallel reductions, simple space complexities do compose, but fail to give precise bounds (this is why we omitted the corresponding corollary). The accurate space complexities $\bar{\sigma}$ reported above were extracted from the complete space–time complexity analysis as the maximum value of $\bar{\gamma}$.

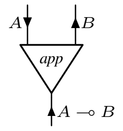
7. Case Study: Higher Order

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 as presented in Section 2. 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).

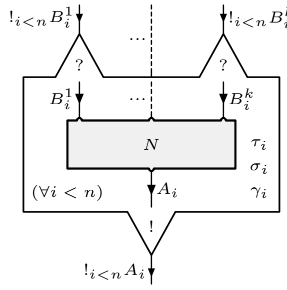
Sized typing schemes and potentials can be assigned as follows. In particular, typing the interface of a box with multisets of size n (they must have the same size) requires typing its contents n times. Requested types for the interface of the contents have been indexed by $i \in [0, n-1]$. Each typing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



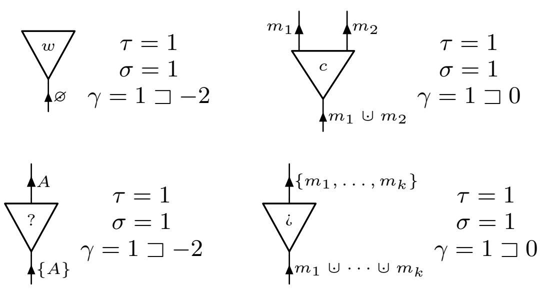
$\tau = 0$
$\sigma = 1$
$\gamma = 1$



$\tau = 1$
$\sigma = 1$
$\gamma = 1 \sqsupset -2$

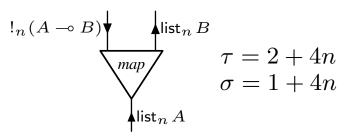


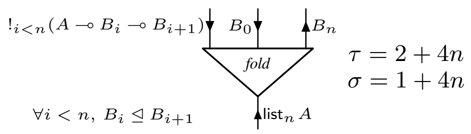
$\tau = \sum_{i<n} \tau_i$
$\sigma = 1 + \sum_{i<n} \sigma_i$
$\gamma = 1 * \coprod_{i<n} \gamma_i$



Multiset inclusion is admissible as subtyping. One can check that with these typing schemes and potentials the related six reduction rules (presented in Section 2) satisfy the potential-decrease property.

One can also verify that the usual *map* and *fold* admit the following typing schemes and potentials:

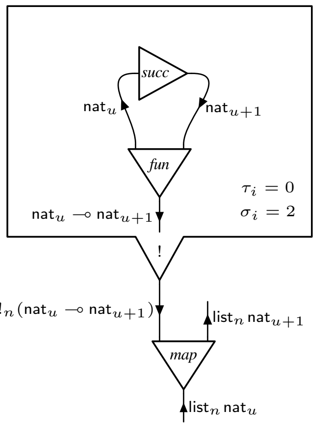




$!_{i<n}(A \multimap B_i \multimap B_{i+1})$
B_0
B_n
fold
$\mathsf{list}_n\, A$
$\forall i < n,\ B_i \trianglelefteq B_{i+1}$
$\tau = 2 + 4n$
$\sigma = 1 + 4n$

Resource usages of functional arguments to *map* and *fold* are taken into account as the potential of the box which holds them. For example:

- Mapping a *succ* : $\mathsf{nat}_p \multimap \mathsf{nat}_{p+1}$ operation with potential $\tau = 0$ and $\sigma = 1$ is done as follows:



succ
nat_u
nat_{u+1}
fun
$\tau_i = 0$
$\sigma_i = 2$
$\mathsf{nat}_u \multimap \mathsf{nat}_{u+1}$
!
$!_n(\mathsf{nat}_u \multimap \mathsf{nat}_{u+1})$
$\mathsf{list}_n\, \mathsf{nat}_{u+1}$
map
$\mathsf{list}_n\, \mathsf{nat}_u$

  This defines an operation of type $\mathsf{list}_n\, \mathsf{nat}_u \multimap \mathsf{list}_n\, \mathsf{nat}_{u+1}$ with a total potential of $\tau = 2 + 4n + \sum_{i<n} 0 = 2 + 4n$ and $\sigma = 1 + 4n + 1 + \sum_{i<n} 2 = 2 + 6n$.

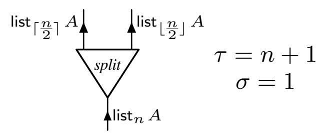
- Folding an *add* : $\mathsf{nat}_p \multimap \mathsf{nat}_q \multimap \mathsf{nat}_{p+q}$ operation with potentials $\tau = 1 + p$ and $\sigma = 1$ demands instantiating A as nat_u and B_i as nat_{v+iu} and defines an operation of type $\mathsf{list}_n\, \mathsf{nat}_u \multimap \mathsf{nat}_v \multimap \mathsf{nat}_{v+nu}$ with potentials $\tau = 2 + 4n + \sum_{i<n}(1+u) = 2 + 5n + nu$ and $\sigma = 1 + 4n + 1 + \sum_{i<n} 3 = 2 + 7n$.

- Similarly, folding a *cat* : $\mathsf{list}_p X \multimap \mathsf{list}_q X \multimap \mathsf{list}_{p+q} X$ operation with potentials $\tau = 1 + p$ and $\sigma = 1$ demands instantiating A as $\mathsf{list}_u X$ and B_i as $\mathsf{list}_{u+iv} X$ and defines a function of type $\mathsf{list}_n(\mathsf{list}_u X) \multimap \mathsf{list}_v X \multimap \mathsf{list}_{v+nu} X$ with potentials $\tau = 2 + 5n + nu$ and $\sigma = 2 + 7n$.

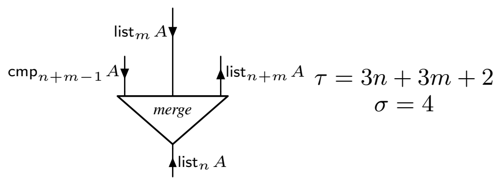
- Folding a *mul* : $\mathsf{nat}_p \multimap \mathsf{nat}_q \multimap \mathsf{nat}_{pq}$ operation with potential $\tau = 2pq + 2p + q + 2$ and $\sigma = pq + p + 1$ demands instantiating A as nat_u and B_i as nat_{vu^i} and defines an operation of type $\mathsf{list}_n\, \mathsf{nat}_u \multimap \mathsf{nat}_v \multimap \mathsf{nat}_{vu^n}$ with potentials $\tau = 2 + 4n + \sum_{i<n}(2vu^{i+1} + 2u + vu^i + 2) = 2 + 6n + 2nu + v(u^n + 2u^{n+1} - 3)$ and $\sigma = 1 + 4n + 1 + \sum_{i<n}(2 + 2vu^{i+1} + 3u + 1) = 2 + 7n + 3nu + v(u^n - 1)$, which are exponential in the size of the list.

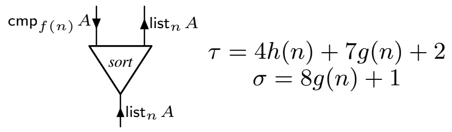
Example of a concrete application to program complexity certification An interaction-net representation of a list of length n containing natural numbers bounded by u has time potential 0 and a space potential bounded by $1 + 2n + nu$. Therefore, the net built by providing this list and an initial value bounded by v (whose time potential is also null and whose space potential is bounded by $v + 1$) as arguments to the net that folds multiplication has combined time potential $\tau = 2 + 6n + 2nu + v(u^n + 2u^{n+1} - 3)$ and combined space potential $\sigma = 4 + 9n + 4nu + vu^n$. According to Corollaries 1 and 2, these combined potentials constitute concrete bounds on the sequential time and space needed to perform this operation as a function of the sizes of its arguments.

An analysis of merge sort Assuming A is replicable, we denote by $\mathsf{cmp}_n A = !_n(A \multimap A \multimap \mathsf{bool})$ the type of comparison functions which can be called n times. Let $f, g, h : \mathbb{N} \to \mathbb{N}$ be recursively defined as $f(n) = n - 1 + f(\lfloor\frac{n}{2}\rfloor) + f(\lceil\frac{n}{2}\rceil)$, $g(n) = 1 + g(\lfloor\frac{n}{2}\rfloor) + g(\lceil\frac{n}{2}\rceil)$ and $h(n) = n + h(\lfloor\frac{n}{2}\rfloor) + h(\lceil\frac{n}{2}\rceil)$ if $n > 1$ or null otherwise. In particular, for $n > 0$, $g(n) = n - 1$ and $f(n) = h(n) - g(n) \leq n \log_2 n$. *Merge sort* can be implemented with the following set of nodes:

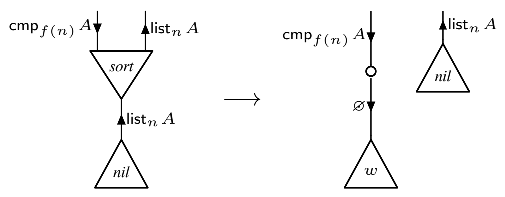


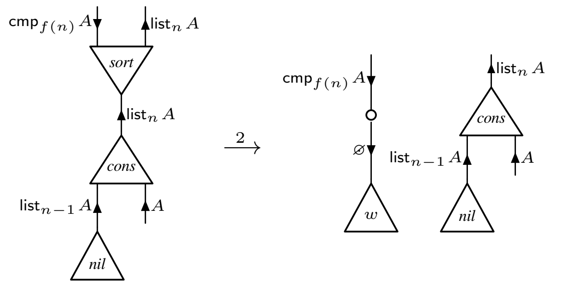
$\mathsf{list}_{\lceil\frac{n}{2}\rceil} A$
$\mathsf{list}_{\lfloor\frac{n}{2}\rfloor} A$
split
$\mathsf{list}_n\, A$
$\tau = n + 1$
$\sigma = 1$

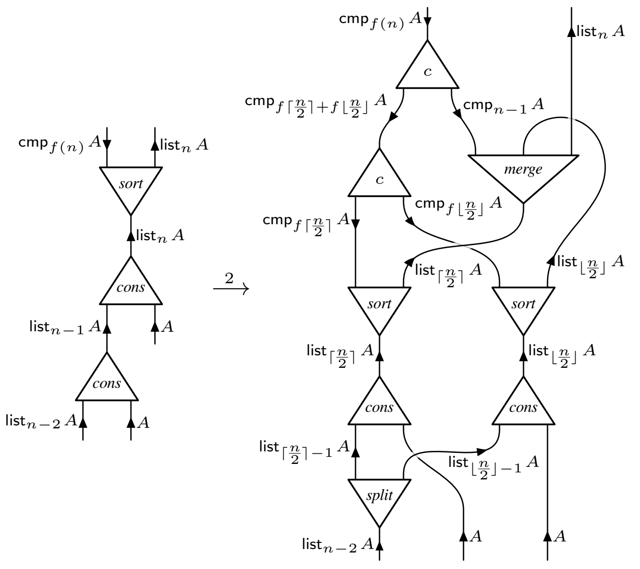




The operations *split* and *merge* are standard. Showing that their implementations satisfy the above typing schemes is not difficult. Reduction rules for *merge sort* are provided below. We display them as nested-pattern reductions [15] to avoid the explicit introduction of auxiliary nodes.







$\mathsf{cmp}_{f(n)}\,A$
$\mathsf{list}_n\,A$
sort
cons
$\mathsf{list}_{n-1}\,A$
$\mathsf{list}_{n-2}\,A$
A
$\xrightarrow{2}$
c
$\mathsf{cmp}_{f\lceil\frac{n}{2}\rceil+f\lfloor\frac{n}{2}\rfloor}\,A$
$\mathsf{cmp}_{n-1}\,A$
merge
$\mathsf{cmp}_{f\lfloor\frac{n}{2}\rfloor}\,A$
$\mathsf{cmp}_{f\lceil\frac{n}{2}\rceil}\,A$
$\mathsf{list}_{\lceil\frac{n}{2}\rceil}\,A$
$\mathsf{list}_{\lfloor\frac{n}{2}\rfloor}\,A$
$\mathsf{list}_{\lceil\frac{n}{2}\rceil-1}\,A$
$\mathsf{list}_{\lfloor\frac{n}{2}\rfloor-1}\,A$
split

Potential-decrease properties follow from the recursive definitions of g and h.

8. Related Work

Complexity Analysis of Interaction Nets Despite the conceptual importance of interaction nets, little is known about complexity analysis of reductions in interaction nets, Perrinel's work [29] on context semantics of interaction nets [13] being the exception. Perrinel employs context semantics to assign a global weight W to a net N which bounds the length of sequential reductions. This is obtained analogously to (and is actually based on) [20]. In contrast to our Corollary 1 this approach is not compositional; it requires an in-depth context semantic analysis of the net N for a given input (equivalent to execution) and does not offer a generic bound.

Implicit Computational Complexity and Program Analysis In implicit computational complexity and in other program-analysis areas of research, space–time complexity functions are seldom used directly. We highlight [4], in which a type-based termination argument is fused together with a semantic size argument in order to classify polytime-computable higher-order functional programs. *Space–time weights* are defined which are conceptually related to the potential functions γ_c. This result is comparable to our Corollary 1, although restricted to polynomial time bounds. On the other hand, no results for space analysis or parallel reduction have been provided, like in our work. With respect to program analysis, we highlight work by Brockschmidt et al. [6] in which an alternating *size* and *time complexity* analysis of integer transition systems is made explicit. Similar ideas are exploited in [31]. The focus is on automation for first-order systems. Automation of time complexity analysis of higher-order programs has for example been considered in [3]. No results for space analysis or parallel reduction have been provided.

Sized Types and Amortized Cost Analysis *Sized types* are a streamlined notion related to *linear dependent types* [21]. In contrast to the analysis provided by Vasconcelos [32] our work is purely focused on user-defined annotations with sized types and additionally employs potentials based on the size annotation, in the spirit of [2]. This is also different from *amortized cost analysis*, where potentials are defined on values and no previous size annotation is required. For example, Hoffmann and Shao [17] provide an extension of earlier multivariate amortized cost analyses [16] to parallel reductions. The costs of parallel executions are essentially approximated using cost-free metrices in addition to a size change analysis. While sized types and amortized cost analysis give rise to powerful automated techniques, the method proposed in this paper is more versatile and allows to incorporate non-local size analysis.

Concurrency and Scheduling Ghica and Smith provide in [8] a generalization of bounded linear logic [12] to bounded linear types over a semiring, which equipped with a suitable SMT solver yields a straightforward automation of type inference. The concept of *comonadic* resource sensitivity is emphasized. It amounts to a shift in paradigm. Instead of precisely witnessing resource use through the type system, as we have advocated here, the focus is on defining scheduling of computational tasks, based on the precise typing.

9. 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. In general, we expect that the present resource analysis will prove useful in the design and implementation of modern hardware solutions.

To conclude, inspired by bounded linear logic [12], 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 [11], 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.

Acknowledgements

We are grateful to the anonymous reviewers whose comments greatly helped us to improve the presentation of the paper. This work was partially supported by FWF (Austrian Science Fund) project number P 25781-N18 and Draper Labs, project number 15-B13.

References

[1] A. Asperti, C. Giovanetti, and A. Naletto. The bologna optimal higher-order machine. *JFP*, 6(6):763–810, 1996.

[2] R. Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 7(2), 2011.

[3] M. Avanzini, U. Dal Lago, and G. Moser. Analysing the complexity of functional programs: Higher-order meets first-order, 2015.

[4] P. Baillot and U. Dal Lago. Higher-order interpretations and program complexity. In *Proc. 21th CSL*, volume 16 of *LIPIcs*, pages 62–76, 2012.

[5] P.v. Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. MIT Press, 1990.

[6] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *Proc. 20th TACAS*, volume 8413 of *LNCS*, pages 140–155, 2014.

[7] M. Fernández, I. Mackie, S. Sato, and M. Walker. Recursive functions with pattern matching in interaction nets. *ENTCS*, 253(4):55–71, 2009.

[8] D. Ghica and A. Smith. Bounded linear types in a resource semiring. In *Proc. 23rd ESOP*, volume 8410 of *LNCS*, pages 331–350, 2014.

[9] S. Gimenez. *Programmer, calculer et raisonner avec les réseaux de la logique linéaire*. PhD thesis, 2009.

[10] S. Gimenez and G. Moser. The complexity of interaction (long version). `http://arxiv.org/abs/1511.01838`.

[11] S. Gimenez and G. Moser. The structure of interaction. In *Proc. 22nd CSL*, volume 23 of *LIPIcs*, pages 316–331, 2013.

[12] J-Y. Girard, A. Scedrov, and P. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *TCS*, 97(1):1–66, 1992.

[13] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proc. 19th POPL*, pages 15–26. ACM Press, 1992.

[14] G. Gonthier, M. Abadi, and J.-J. Lévy. Linear logic without boxes. In *Proc. 7th LICS*, pages 223–234. IEEE, 1992.

[15] A. Hassan and S. Sato. Interaction nets with nested pattern matching. *ENTCS*, 203(1):79–92, 2008.

[16] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *TOPLAS*, 34(3):14, 2012.

[17] J. Hoffmann and Z. Shao. Automatic static cost analysis for parallel programs. In *Proc. 24th ESOP*, volume 9032 of *LNCS*, pages 132–157, 2015.

[18] Y. Lafont. Interaction nets. In *Proc. 17th POPL*, pages 95–108. ACM, 1990.

[19] Y. Lafont. Interaction combinators. *IC*, 137(1):69–101, 1997.

[20] U. Dal Lago. Context semantics, linear logic, and computational complexity. *TOCL*, 10(4), 2009.

[21] U. Dal Lago and B. Petit. Linear dependent types in a call-by-value scenario. *Sci. Comput. Program.*, 84:77–100, 2014.

[22] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. 7th POPL*, pages 16–30. ACM Press, 1990.

[23] S. Lippi. Universal hard interaction for clockless computation: Dem Glücklichen schlägt keine Stunde! *FI*, 91(2):357–394, 2009.

[24] The linear logic wiki: Fragments. `http://llwiki.ens-lyon.fr/mediawiki/index.php/Fragment`.

[25] I. Mackie. Interaction nets for linear logic. *TCS*, 247(1-2):83–140, 2000.

[26] I. Mackie. Efficient lambda-evaluation with interaction nets. In *Proc. 15th RTA*, volume 3091 of *LNCS*, pages 155–169, 2004.

[27] D. Mazza. *Interaction Nets: Semantics and Concurrent Extensions*. PhD thesis, 2006.

[28] P.-A. Mellies. Functorial boxes in string diagrams. In *Proc. 15th CSL*, LNCS, pages 1–30, 2006.

[29] M. Perrinel. On context semantics and interaction nets. In *Proc. Joint 23rd CSL and 29th LICS*, page 73. ACM, 2014.

[30] J. S. Pinto. Parallel evaluation of interaction nets with mpine. In *Proc. 12th RTA*, volume 2051 of *LNCS*, pages 353–356, 2001.

[31] M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Proc. 26th CAV*, volume 8559 of *LNCS*, pages 745–761, 2014.

[32] P. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St. Andrews, 2008.

[33] L. Vaux. *λ-calcul différentiel et logique classique: interactions combinatoires*. PhD thesis, 2007.