A Programming Language for Interaction Nets (Extended Abstract)

Marc Thatcher
University of Sussex
Falmer, U.K.
m.thatcher@sussex.ac.uk

We present *function-constructor nets*, a Turing-complete subset of interaction nets in which agents are partitioned into *functions* and *constructors*, inducing a notion of observable output whilst preserving the permutation equivalence of reduction sequences and hence the potential for parallel execution. We then introduce FLIN, a language with a functional-style syntax for programming such nets, and describe several extensions to the core language to improve expressiveness and practicality. Finally, we present a prototype implementation that demonstrates the viability of the approach.

1 Introduction

Interaction nets [13] are a generalisation of Girard's *proof structures* [9] and have been considered both as a formal computational model [10] and a form of programming language [8, 17]. As a programming language, they have several desirable properties: memory management is syntactically enforced, data sharing is natural, they are Turing complete and, perhaps most importantly, evaluation is inherently parallel. Parallel execution has become of increasing importance in the decades since the original paper was published, as multi-core and multi-threaded processor architectures have become the norm. But mainstream programming languages have not kept up: whether imperative or declarative it is usually left up to the programmer to explicitly annotate which parts of the program can be run in parallel, and which must be sequential.

Interaction nets free us from such annotations. Because net reduction sequences are *permutation equivalent* (see paragraph **Rewriting**), graph reduction can be executed in any order, or in parallel, without any programmer intervention. This is not merely a theoretical observation: the programming language INPLA [25] provides empirical evidence for the improvements available from nets' parallel execution, with near-linear speedups as more processing power is brought on-line, and benchmark running times beating well-established industrial-strength languages such as `Haskell`, `OCaml`, `SML` and `Python`. The practical case for interaction nets as an evaluation mechanism is therefore strong; what has been missing is a programming language that makes them accessible.

Each of the existing languages that use interaction nets as their underlying evaluation mechanism occupies a different point in the design space. INPLAprovides a text syntax close to the underlying formalism, well-suited for expressing nets directly, but which requires programmers to think at the level of agents, nets and rewrite rules rather than at the level of functions and data. In contrast, HVM [3] and VINE [27] are imperative languages, based on `Python` and `Rust` respectively, which compile to interaction nets for evaluation, making them more accessible to programmers but introducing a compilation layer between the source-level operational semantics and the net-level evaluation, with the correspondence between the two established by construction rather than through explicit formal results.

The aim of this paper is two-fold. The first is foundational: despite the attractive properties of interaction nets for programming listed above, they do not provide a notion of observable value [20].

Submitted to:
Termgraph 2026

Section 3 considers this, and to address it, defines *function-constructor nets* which partition net nodes into computational *functions* and data-bearing *constructors*. Together these define input and output of nets and hence observable value. This section is a summary of Section 3.3 of the author's PhD thesis [28]. The second is practical. Section 4 presents FLIN, a language for programming function-constructor nets, based on Section 3.4 of [28]. We then describe a number of extensions that draw together several existing strands of work to form a more complete programming system. Sources are cited throughout, and correctness proofs are omitted in favour of the original references. Section 5 outlines, and points to, our prototype implementation.

2 Interaction nets

Interaction nets: An interaction net is a finite, labelled, undirected, possibly cyclic, possibly disconnected, graph composed of zero or more nodes (*agents*) and edges (*wires*). Agents are labelled by symbols from a finite *signature*, Σ. Each symbol has an associated *arity*, given by $ar : \Sigma \to \mathbb{N}$. An agent labelled with a symbol of arity n has $n+1$ ports: one distinguished *principal* port and n *auxiliary* ports. When convenient, ports may be labelled. A port on an agent may be the end-point of at most one wire; wires connect two ports together. Any port that can have a wire connected to it is said to be *free*. In particular, a wire can connect two free ports; see for example the wire between ports h and r in Figure 3. We comment that the *empty net*, containing neither agents nor wires, is an interaction net.

Figure 1 shows a number of example agents, where principal ports are drawn as grey triangles and auxiliary ports as grey circles. An example net (with one free port, r) is given in Figure 2, drawn using the conventional proof-structure orientation.

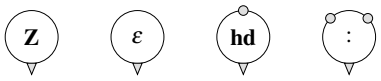


Figure 1: Example agents

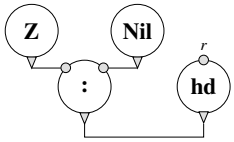


Figure 2: Example interaction net

Nets are rewritten by applying *interaction rules*.

Interaction rules: Two agents connected by their principal ports form an *active pair*. An interaction rule is a pair of nets written $LHS \Rightarrow RHS$, where:

- *LHS* is an active pair. Its set of free ports is its *interface*.
- *RHS* is a net with the same interface as *LHS*.

Each active pair has at most one associated interaction rule. An example rule is Figure 3; observe that both sides have the same interface, the set of ports labelled h, t and r. Such interface invariance requires

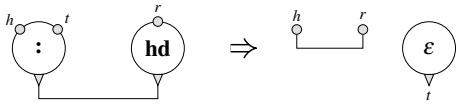


Figure 3: Example interaction rule

explicit erasure of any connection at t. In Figure 3 this is done using the ε agent (defined in Section 4.1).

Given a net and a set of rules, we can evaluate the former by applying the latter.

Rewriting: Applying a single rule to a net follows standard term graph rewriting [23], without automatic garbage collection, whereby the active pair matching the *LHS* of a rule is replaced by its *RHS*, suitably connected to the surrounding net. An example of two sequential rewrites is shown in Figure 4.

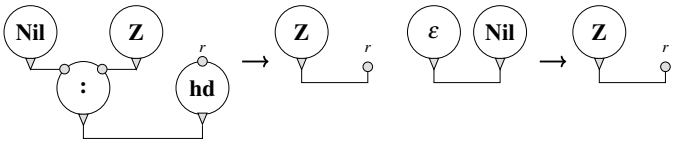


Figure 4: Example net reduction

As rewriting replaces two agents, it is *local*. As it is also deterministic, net rewriting is *permutation equivalent*[1]: if there is more than one active pair in a net, rewriting all of them results in the same net, no matter what order is chosen. Consequently, all active pairs may be reduced *in parallel* and we will assume this in the sequel.

Evaluation: Net evaluation is defined as repeated rewriting until a net without active pairs—a *normal form*— is reached. Normal forms are unique [19, §1.2.1] but may not exist [13, Fig. 3].

3 Function-Constructor Nets

Despite the advantages listed in Section 1, programming with interaction nets raises both foundational and practical challenges. The foundational issues arise from the identification of results with normal forms, which we address in this section. The practical problems are syntax and usability. These are considered in Section 4.

To understand the problem of normal forms as results, consider Figures 5 to 7. Each is a normal form, but the first two are *inaccessible*—they cannot be composed with other nets—and whilst the third, (a *vicious circle*[2]), can be, as its free port is an auxiliary one, information cannot be extracted from it. It may therefore be considered a *meaningless* net. Conversely, nets without normal forms but which constantly produce accessible agents, such as *streams*, have computational meaning but are not considered to have a result.

Figure 5: The empty net

Figure 6: A closed net

Figure 7: A vicious circle

To address this, we give a new definition of the result of a net evaluation. To start, we refine ports by distinguishing *inputs* from *outputs* and constraining wires to be between an input and an output. An agent whose principal port is an input is a *function*; its auxiliary ports may be either inputs or outputs. An

[1] Often called *strong confluence* following [13]. A discussion of nomenclature and our preference for "permutation equivalent" can be found in [28, §2.5.1].

[2] So named in [13].

agent whose principal port is an output is a *constructor*; all its auxiliary ports are inputs[3]. This induces a partition of the signature, Σ, into disjoint subsets, Σ_F and Σ_C, of function and constructor symbols respectively.

A *function-constructor net* then is one with symbols drawn from the Σ_F and Σ_C sub-signatures. A net's interface is now split into non-overlapping input- and output-interfaces. Interaction rule's *LHS* must be between a function and a constructor, and the *RHS* must be a function-constructor net. Such nets are strict subsets of interaction nets but retain their computational power: for example the Turing complete *directed combinators* [14, §3.4] are function-constructor nets.

We define the *result* of evaluating a function-constructor net not as its normal form, but as the coinductively defined constructor tree observed at an output port, where each finite prefix is produced after finitely many interaction steps. Under this definition, none of Figures 5 to 7 have results, but a net which generates an infinite stream output does.

This definition of function-constructor nets is adapted from [28, §3.3]; interaction net productivity is defined in [28, Chap.5]. The implications for defining an alternative notion of result are explored further in [29].

Prior work: The idea of dividing agents into two groups is not novel; a similar proposal is in [13, §2][4], [5] and [7]. However these approaches are are formulated differently, allowing constructors with multiple outputs and not constraining wires to be between inputs and outputs. The term "productivity" was coined by Dijkstra [4] and has been extensively considered in term rewriting but not previously for interaction nets. Replacing normal forms with productive outputs is also novel, although there is a foreshadowing of this in the definition of *observable interface* in [5].

4 FLIN: a Functional Language for Interaction Nets

A number of programming languages for interaction nets have been defined, starting with [13, §3], including [1, 8, 11, 15, 21], culminating with INPLA [24][5]. These systems differ in emphasis, but they share a common feature: nets and rules are expressed *explicitly* as textual representations of agents and wires. This low-level style departs significantly from the abstractions and syntax found in mainstream programming languages. For example, one INPLA definition of the rule shown in Figure 3 is[6]: `head(r)><Cons(h,t) => r~h, erase(t);`, which follows the proposed syntax of [17, §4].

In order to allow direct interaction net programming (unlike the indirect, compiled approaches of HVM and VINE) but with a more familiar syntax, we build upon the function-constructor net model. FLIN terms represent nets and are defined as the smallest set generated by the following grammar[7] in which each port label is used exactly once:

$$t, u \coloneqq \texttt{-} \mid x \mid f(t^+) \mid C(t^*) \mid \texttt{let}\ [x^*] \sim t\ \texttt{in}\ u \mid t \mid u$$

- $-$ represents the empty net.

[3]Restricting a constructor to having a single output is a design decision. Constructors represent atomic data elements, thus once consumed in an interaction there should not be a residual wire or agent.

[4]There called *destructors* and *constructors*.

[5]There are also more general graph rewriting languages which could be used to encode interaction nets, such as `LMNtal` (`www.uedalab.jp/lmntal`) but these provide more expressiveness, and hence complexity, than we need.

[6]This is one of eight ($\alpha-$)equivalent ways of writing this rule: the order of agents on the *LHS* can be switched, the `r` and `h` port ordering can be reversed and the order of the two *RHS* terms can be exchanged.

[7]See [28, §3.9] for a more detailed description of the syntax, its rationale, and further examples.

- x denotes a wire whose input port is labelled x and whose output port is unlabelled. We will use single lower-case letters as port labels, sometimes with numeric suffixes.
- $f(t^+)$ is a function term composed of a function agent labelled $f \in \Sigma_F$ with one or more terms as inputs and unlabelled outputs. Alphanumeric strings with initial lower case letters will be used as function labels.
- $C(t^*)$ is a constructor term composed of a constructor agent labelled $C \in \Sigma_C$ with zero or more terms as inputs, and an unlabelled output, written as an initially upper case alphanumeric string.
- `let` $[x^*] \sim t$ `in` u names the output interface of t with the list of port variables, $[x^*]$. As $[x^*]$ only provides names, it is not a port label usage as restricted in the syntax. These labels may be used in t and/or u and there imply wires; see Figure 8 for an example. Within such a term the binding acts as `let` $([x^*] = t)$ `in` (u).
- $t \mid u$ is a *juxtaposed* term: this represents two disjoint subnets defined by t and u, such as the middle net in Figure 4. Note that $t \mid -$ is equivalent to t.

We now set an orientation convention: agents are rotated so that their input ports are below their label, and outputs, above. If agent α has an input port connected to an output of β, we draw α above β. For example, in Figure 8, the `S` agents' principal ports, and the two auxiliary ports of δ (labelled x_1 and x_2) are outputs; the `S` agents' auxiliary ports, and δ's principal port (labelled x) are inputs.

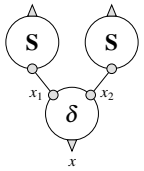


Figure 8: Net described by `let[x1,x2]~delta(x) in S(x1) | S(x2)`

Given such a *LHS* and *RHS*, we define the rule syntax as $LHS = RHS$, and refer to such rules as *function definitions*. Proofs of the equivalence of terms and nets, and function definitions and rules can be found in [28, §3].

We give two simple definitions, for unary number addition and multiplication, that we will use later:

```
add(Z,y)    = y
add(S(x),y) = S(add(x,y))
mult(Z,y)    = Z | erase(y)
mult(S(x),y) = let[y1,y2]~dup(y) in add(y1,mult(x,y2))
```

Given a set of definitions, a FLIN *program* is a term without any free input ports.

Remark: Despite its functional syntax, FLIN's semantics matches that of function-constructor nets [28, §3.7].

The remainder of this section considers how to improve FLIN's expressiveness and ease of use.

4.1 Generic constructors

As the *LHS*s of rules are defined as two agents, any function that acts in the same way on different inputs needs to be defined multiple times. For example, defining erasure when there are 0-arity constructors `Z` for numbers, `Nil` for lists and `Empty` for trees requires

```
erase(Z) = -        erase(Nil) = -        erase(Empty) = -
```

Each of these could be generated at compile time by a single definition where a meta-constructor is used to stand in for each constructor above:

```
erase(*) = -
```

where `*` refers to any 0-arity constructor. We call these *generic* constructors. Such generic constructors allow polymorphic function definitions. We can then use `*(x)`, `*(x,y)`, etc. for constructors of higher arity: `erase(*(x)) = erase(x)`, `erase(*(x,y)) = erase(x)|erase(y)`, etc.

Duplication can be defined similarly as:

```
dup(*)      = * | *
dup(*(x))   = let [x1,x2] ~ dup(x) in *(x1)|*(x2)
dup(*(x,y)) = let [x1,x2] ~ dup(x) in let [y1,y2] ~ dup(y) in *(x1,y1)|*(x2,y2)
```

Prior work: The formal notion of generic constructors is novel [28, §3.4], although an informal notion of writing a variable in place of an agent symbol has been used, such as in [12, §4]). *Templates* [17] are the closest to our definition, but are a more general macro system.

4.2 Implicit memory management

Interaction nets' explicit memory management has many advantages—no separate garbage collection, precise control of resources, lower overhead and simpler implementation—but it is burdensome for the programmer to continually have to insert `dup`s and `erase`s, and the programs quickly become difficult to read.

Consider unary number multiplication from Section 4: dropping the `erase` in the base case and `dup` in the successor case is syntactically invalid, but more natural for the programmer:

```
mult(Z,y)    = Z
mult(S(x),y) = add(y,mult(x,y))
```

Such definitions can be systematically translated into well-formed FLIN by the following transformation:

- For each variable x which occurs on the *LHS* but not the *RHS*, add "`| erase(`x`)`" to the *RHS*.
- For each variable x which occurs n times ($n > 1$) on the *RHS*, add "`let[`x_{n-1}`,`x_n`]=dup(`x_{n-2}`) in (let[`x_{n-3}`,`x_{n-2}`]=...dup(`x`)...)` to the start of the *RHS*.

This translation is deterministic up to associativity of duplication and yields a valid function-constructor net. We are not aware of this formulation having been made explicit in this form in prior work.

4.3 Nested pattern matching

Another limitation of interaction rules is that they only match the labels on agents in active pairs, and therefore can only inspect the outermost structure of terms. This makes definitions involving nested pattern matching cumbersome, as deeper structure must be exposed incrementally.

For example, consider the function returning the last element of a list:

```
last([x])      = x
last(x1:x2:xs) = last(x2:xs)
```

where list notation is syntactic sugar and we use implicit memory management in the second definition. Both definitions use nested pattern matching. This is evident when the corresponding constructor structure is made explicit: the first is `Cons(x,Nil)` requiring a match on the `Nil` below the `Cons`, and the second, `Cons(x1,Cons(x2,Nil))`, needs to access both the second `Cons` and the `Nil`.

In [12], a semantics-preserving translation is given from such nested pattern matching definitions to standard interaction nets subject to rules being sequential and non-overlapping. We extend FLIN to allow such rules.

4.4 Natural numbers

Section 4.2 highlights another issue: interaction nets do not have an intrinsic notion of numbers. Following the approach of PCF [22], we add natural number agents by extending a finite signature Σ to an infinite one, $\Sigma_{\mathbb{N}} = \Sigma \cup \mathbb{N}$, and adding successor and predecessor definition schemata: `succ(_n) = _n + 1`, `pred(_n) = _n - 1`, where `_n` indicates that this is not a port name, but rather an agent label ranging over $\mathbb{N}$, and $n-1$ is defined to be *monus*, i.e. with a floor at 0. Natural number addition and multiplication can be defined (with implicit memory management) as:

```
add(0, n)  = n
add(_m,_n) = succ(add(pred(m),n))
mult(0,n)   = 0
mult(_m,_n) = add(mult(pred(m),n),n)
```

Prior work: This has been presented informally, such as in [18]. INPLA uses a system based on *attributes* [6], which are a more general notion providing a form of foreign function interface.

4.5 Multiple principal ports

Consider a function that returns the maximum of two unary numbers. We would like to write this as:

```
max(Z,y)       = y
max(S(x),Z)    = S(x)
max(S(x),S(y)) = S(max(x,y))
```

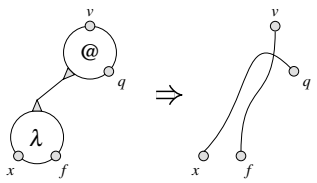
but this pattern matches on both inputs, not just the principal port. However, when pattern matching is sequential (i.e. arguments can be considered in a fixed order), such rules can be compiled into interaction nets preserving the semantics by introducing auxiliary agents that sequentialise the matching process [26]. We include this in FLIN.

The syntax is therefore extended to allow function agents to possess any strictly positive integer number of principal ports. We require that all principal ports participate in active pairs on the left-hand side, so that no principal ports remain exposed in the interface of a rule. In such systems, permutation equivalence of reduction is preserved [26, Proposition 4.3].

Rules that are fundamentally non-sequential, such as *parallel-or* [22] or Berry's non-sequential system [2], cannot be compiled away in this manner.

4.6 Higher-order functions

A drawback of FLIN, at least from the perspective of a functional programmer, is that it is a first-order system: we cannot, for example, write `map (add 1) [1,2,3]` as `map`'s and `add`'s principal ports do not connect. To address this, we follow the approach of [16] and add a *higher-order constructor* symbol λ, with two output ports and an application function symbol, @, both of arity-2, with one rule:



This λ agent does not fit into our function/constructor dichotomy and therefore does not occur within FLIN terms, but arises when FLIN is compiled to INPLA. The recursive case of `map` can now be defined as `map((x:xs),^f) = ((^f x):(map(xs,*f))`, where the `^f` indicates that this is a function argument. This is compiled to[8] Figure 9, where the @ and δ_f agents are induced by the `^f` syntax, and an example use is shown in Figure 10. This duplication agent, δ_f, is non-functional as eventually it will need to

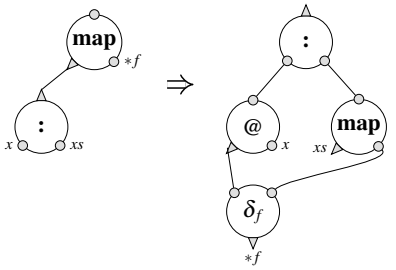


Figure 9: `map` rule

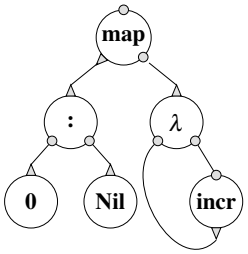


Figure 10: `map` applied to `incr` and `[0]`

interact with another δ_f agent. For this reason we keep these agents "under the hood", available to the compiler but not the programmer.

5 Implementation

A prototype FLIN-to-INPLA compiler is available at `https:github.com/MarcThatcher/Termgraph-implementation` with instructions for use and a number of examples.

We give one example here: the n-th Fibonacci number can be calculated in FLIN, using the natural number and implicit memory management features, as:

```
fib(0)  = 0
fib(1)  = 1
fib(_n) = add(fib(pred(n)), fib(pred(pred(n))))
```

This is compiled to:

```
fib(r) >< (int n) | n==0 => 0~r | n==1 => 1~r | _ => Dup(n1,n2)~n, pred(r_0r_1_1_1)~n2,
                    pred(r_0r_1_1)~r_0r_1_1_1, fib(r_0r_1)~r_0r_1_1, pred(r_0_1)~n1,
                    fib(r_0)~r_0_1, add(r,r_0r_1)~r_0;
```

Acknowledgements

The author thanks Ian Mackie for suggesting this topic and his comments on an earlier draft, and the reviewers for their valuable comments and suggestions.

References

[1] José Bacelar Almeida, Jorge Sousa Pinto & Miguel Vilaça (2008): *A Tool for Programming with Interaction Nets*. *Electronic Notes in Theoretical Computer Science* 219, pp. 83–96, doi:10.1016/j.entcs.2008.10.036.

[8] This definition reverses the usual order of inputs; an extra rule can be included to deal with this if wanted.

[2] G. Berry (1976): *Bottom-up computation of recursive programs*. Revue Française d'Automatique, Informatique et Recherche Opérationnelle 10(3), pp. 47–82.

[3] Higher Order Company (2025): *HigherOrderCo*. `https://higherorderco.com/`. Accessed: 2026-01-08.

[4] Edsger W. Dijkstra (1980): *On the productivity of recursive definitions (EWD749)*. EWD749.

[5] Maribel Fernández & Ian Mackie (1999): *A Calculus for Interaction Nets*. In: Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 170–187, doi:10.1007/10704567_10.

[6] Maribel Fernández, Ian Mackie & Jorge Sousa Pinto (2001): *Combining interaction nets with externally defined programs*. In Luís Moniz Pereira & Paulo Quaresma, editors: *APPIA-GULP-PRODE 2001: Joint Conference on Declarative Programming, Évora, Portgual, September 26-28, 2001, Proceedings, Évora, Portugal, September 26-28, 2001*, Departamento de Informática, Universidade de Évora, pp. 297–312. Available at `http://www.di.uevora.pt/%7Epq/agp01/finals/19.pdf`.

[7] M. Ferndandez & I. Mackie (1998): *Coinductive techniques for operational equivalence of interaction nets*. In: Proceedings. Thirteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.98CB36226), LICS-98, IEEE Comput. Soc, doi:10.1109/lics.1998.705668.

[8] Simon Gay (1991): *Interaction Nets*. Master's thesis, Cambridge. Diploma in Computer Science thesis.

[9] Jean-Yves Girard (1987): *Linear logic*. Theoretical Computer Science 50(1), pp. 1–101, doi:10.1016/0304-3975(87)90045-4.

[10] Georges Gonthier, Martín Abadi & Jean-Jacques Lévy (1992): *The Geometry of Optimal Lambda Reduction*. In: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '92, ACM Press, pp. 15–26, doi:10.1145/143165.143172.

[11] Abubakar Hassan, Ian Mackie & Shinya Sato (2008): *Interaction nets: programming language design and implementation*. Electronic Communications of the EASST, p. Volume 10: Graph Transformation and Visual Modeling Techniques 2008, doi:10.14279/TUJ.ECEASST.10.156.

[12] Abubakar Hassan & Shinya Sato (2008): *Interaction Nets With Nested Pattern Matching*. Electronic Notes in Theoretical Computer Science 203(1), pp. 79–92, doi:10.1016/j.entcs.2008.03.035.

[13] Yves Lafont (1990): *Interaction Nets*. In: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '90, ACM Press, pp. 95–108, doi:10.1145/96709.96718.

[14] Yves Lafont (1997): *Interaction Combinators*. Information and Computation 137(1), pp. 69–101, doi:10.1006/inco.1997.2643.

[15] Sylvain Lippi (2002): *in2: A Graphical Interpreter for Interaction Nets*. In: Rewriting Techniques and Applications, Springer Berlin Heidelberg, pp. 380–385, doi:10.1007/3-540-45610-4_29.

[16] Ian Mackie (1998): *YALE - Yet Another Lambda Evaluator based on interaction nets*. ACM SIGPLAN Notices 34(1), pp. 117–128, doi:10.1145/291251.289434.

[17] Ian Mackie (2005): *Towards a Programming Language for Interaction Nets*. Electronic Notes in Theoretical Computer Science 127(5), pp. 133–151, doi:10.1016/j.entcs.2005.02.015.

[18] Ian Mackie (2010): *A Visual Model of Computation*. In: Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 350–360, doi:10.1007/978-3-642-13562-0_32.

[19] Damiano Mazza (2006): *Interaction nets: Semantics and Concurrent Extensions*. Ph.D. thesis, Université Aix-Marseille II/Universita degli Studi Roma Tre.

[20] James Hiram Morris (1969): *Lambda-calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology. Available at `http://hdl.handle.net/1721.1/64850`.

[21] J. S Pinto (2001): *Parallel Implementation with Linear Logic*. Ph.D. thesis, Ecole Polytechnique.

[22] G.D. Plotkin (1977): *LCF considered as a programming language*. Theoretical Computer Science 5(3), pp. 223–255, doi:10.1016/0304-3975(77)90044-5.

[23] Detlef Plump (1999): *Term Graph Rewriting*. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors: *Handbook of Graph Grammars and Computing by Graph Transformation*, 2, World Scientific, Singapore, pp. 3–61.

[24] Shinya Sato (2014): *Design and implementation of a low-level language for interaction nets*. Ph.D. thesis, University of Sussex.

[25] Shinya Sato (2025): *Inpla: Interaction nets as a programming language*. `https://github.com/inpla/inpla`. Accessed: 2026-01-23.

[26] François-Régis Sinot & Ian Mackie (2005): *Macros for Interaction Nets*. *Electronic Notes in Theoretical Computer Science* 127(5), pp. 153–169, doi:10.1016/j.entcs.2005.02.016.

[27] T6 (2025): *The Vine Programming Language*. `https://vine.dev/`. Accessed: 2026-01-23.

[28] Marc Thatcher (2025): *Designing a High-Level Programming Language for Interaction Nets*. Ph.D. thesis, University of Sussex. PhD thesis.

[29] Marc Thatcher (2026): *Computing with Infinite Interaction Nets (Presentation)*. Presented at the Continuity, Computability, Constructivity Workshop May 2026 `https://marcthatcher.github.io/talks.html`. Accessed: 31-05-2026.