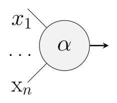
A Functional Programming Language for Interaction Nets – Abstract

Marc Thatcher
Department of Informatics, University of Sussex,
Falmer, Brighton BN1 9QJ, UK
m.thatcher@sussex.ac.uk

29th September, 2023

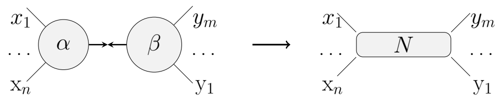
Interaction nets, introduced by Yves Lafont in [1], are a form of graph rewrite system, which can be considered as a low-level programming language or a model of computation.

Programs and data are represented as nets, composed of nodes (*agents*) each of which has one or more *ports* which allow agents to be connected via edges (*wires*). The one mandatory port is called the *principal port* with any others being *auxiliary ports*. Agents are drawn as follows, with the principal port indicated by an arrow:



Ports which are not connected to agents are called *free ports*.

Evaluation occurs by applying rewrite or *interaction rules*. These can be defined for any two agents connected by their principal ports and the rewrite is to any valid net which maintains the set of free ports. So in general we have:



Note that the requirement that the set of free ports is invariant under the rewrite rule means that rewrites are *linear* – i.e. any variable on the left-hand side must appear exactly once on the right-hand side. For the logic underlying this, see [2].

Such nets have a number of attractive features as a programming language, including:

- They are Turing complete.
- Locality of interaction rules means reductions can proceed in parallel. It is not necessary to rewrite programs to take advantage of extra processors or cores.
- Memory management, including garbage collection and copying, is enforced by the linearity requirement and works as well in parallel as sequentially.
- Algorithms are encoded directly onto data structures without the usual "scaffolding" of programming languages.
- Their visual nature can aid program writing, debugging and learning.

They therefore address a key issue facing functional programming languages – how to include concurrency. Current solutions require new libraries, forking, differentiating between parallelism and concurrency – all of which get between the programmer and the program ; the very thing that functional programming is supposed to help with!

Although Lafont introduced interaction nets as "a new kind of programming language", research has concentrated on using them as a computational model to consider, *inter alia*, efficient lambda calculus evaluation (see for example [3]) and there has been relatively little work done on exploring how the language aspect could be practically implemented. The textual languages that have been suggested have been limited to explicit descriptions of the diagrams, leading to unwieldy and unusual looking syntax. For example, Haskell's
`append (x:xs) ys = x:(append xs ys)` becomes
`Cons[x,Append(v,t)]><Append[v,Cons(x,t)]`.

It is the authors' contention that a functional-style language that can be used to directly define interaction nets and rules would provide great utility to the PL community. Such a language could be used directly, or more likely, as an intermediate language sitting below a fuller-featured language such as Haskell or OCaml. The presentation will describe the work being done on such a language, currently titled FLIN for "Functional Language for Interaction Nets"[1] including:

- How to map function definitions to interaction nets and vice versa. Limits to this and how we overcome them.
- How garbage collection and data copying work.
- Extensions to pure interaction nets to aid expressiveness.
- Higher order functions and monads.
- Type systems.
- Empirical data on the performance benefits available from parallelism.

[1]One positive outcome of the meeting would be a better name!

References

[1] Yves Lafont, *Interaction Nets* Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL, 1990.

[2] Jean-Yves Girard *Linear Logic* Theoretical Computer Science, Volume 50, 1987.

[3] Ian Mackie *YALE: Yet Another Lambda Evaluator Based on Interaction Nets* Association for Computing Machinery (ACM), Volume 34, 1998.