Visual Programming with Interaction Nets

Abubakar Hassan, Ian Mackie, and Jorge Sousa Pinto

[1] Department of Informatics, University of Sussex, Falmer, Brighton BN1 9QJ, UK
[2] LIX, École Polytechnique, 91128 Palaiseau Cedex, France
[3] Departamento de Informática/CCTC, Universidade do Minho, Braga, Portugal

Abstract. Programming directly with diagrams offers potential advantages such as visual intuitions, identification of errors (debugging), and insight into the dynamics of the algorithm. The purpose of this paper is to put forward one particular graphical formalism, interaction nets, as a candidate for visual programming which has not only all the desired properties that one would expect, but also has other benefits as a language, for instance sharing computation.

1 Introduction

Interaction nets were introduced in 1990 [8]. The theory and practice of interaction nets have been developed over the last years to provide tools to program and reason about interaction net programs. For instance, a calculus [3], notions of operational equivalence [4], encodings giving implementations of rewriting systems such as term rewriting systems [2] and the lambda calculus [5, 9]. However the visual programing aspect of this language has been neglected. The purpose of this paper is to take a fresh look at this formalism from a visual programming perspective and demonstrate that it is a suitable language to develop programs for the following reasons:

- It is a diagrammatic programming language where both programs and data are given the same status: they are both given by diagrams in the same formalism.
- Computation is rule based: the programmer explains the algorithm in terms of rules which are applied by the runtime system. From this perspective, it could be classified as a declarative language.
- All the computation is expressed by the rules: there are no external mechanisms performing parts of the computation, and consequently the diagrams give a full description.
- The rewrite rules transform the diagrams, and this gives a trace of the computation directly at each step. An application to algorithm animation would be a by-product of the approach.

Interaction nets are not new. They have been applied very successfully to representing sharing in computation. In the present paper this is not a feature that we particularly want to develop, but nevertheless it is a convenient plus

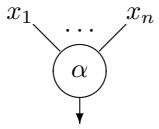
point that we will come back to later. The purpose of this paper is to show, through examples, that interaction nets are not only a good example of a visual programming language, but they have features that could make them a viable programming paradigm when appropriate tools are developed.

Work closest to ours is in the area of Visual Functional Programming which has addressed different aspects of visual programming. The Pivotal project [6] offers a visual notation (and Haskell programming environment) for data-structures, not programs. Visual Haskell [12] more or less stands at the opposite side of the spectrum of possibilities: this is a dataflow-style visual notation for Haskell programs, which allows programmers to *define* their programs visually (with the assistance of a tool) and then have them translated automatically to Haskell code. Kelso's VFP system [7] is a complete environment that allows to define functional programs visually and then reduce them step by step. Finally, VisualLambda [1] is a formalism based on graph-rewriting: programs are defined as graphs whose reduction mimics the execution of a functional program. As far as we know none of these systems is widely used.

In the next section we introduce the formalism. We then give two examples of the use of this language. We conclude the paper with a discussion about other features of interaction nets, including parallelism, sharing, and perspectives for use as programming language in the larger scale (hierarchical nets).

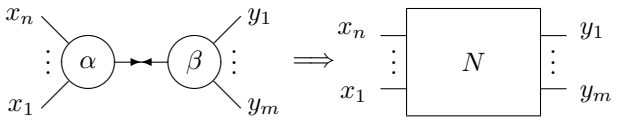
2 Interaction Nets

We begin by defining the graphical rewriting system, which is a generalisation of interaction nets found in the literature (see [8] for instance). We have a set Σ of *symbols*, which are names of the nodes in our diagrams. Each symbol has an arity ar that determines the number of *auxiliary ports* that the node has. If $ar(\alpha) = n$ for $\alpha \in \Sigma$, then α has $n + 1$ *ports:* n auxiliary ports and a distinguished one called the *principal port*.



Nodes are drawn variably as circles, triangles or squares, and they optionally have an attribute, which is a value of base type: integers and booleans. We write the attribute in brackets after the name: e.g. $c(2)$ is a node called c which holds the value 2. A *net* built on Σ is an undirected graph with nodes at the vertices. The edges of the net connect nodes together at the ports such that there is only one edge at every port. A port which is not connected is called a *free port*.

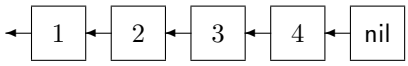
Two nodes $(\alpha, \beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair*, which is the interaction nets analogue of a reducible expression (redex). A rule $((\alpha, \beta) \Longrightarrow N)$ replaces the pair (α, β) by the net N. All the free ports are preserved during reduction, and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.



If either (or both) of the nodes are holding a value, then we can use these values to give different right-hand sides of the rule by labelling the arrow with a condition. The condition can be built out of the usual boolean operators ($<$, $>$, $=$, $!=$, etc.). However, the conditions must be all disjoint (there cannot be two rules which can be applied). Each alternative must of course give a net satisfying the property given above for the case without attributes. The most powerful property that this system has is that it is one-step confluent: the order of rewriting is not important, and all sequences of rewrites are of the same length and equal (in fact they are permutations). This has practical aspects: the diagrammatic transformations can be applied in any order, or even in parallel, to give the correct answer.

We next explain how to represent a simple data structure using interaction nets. In the next section we give two examples of algorithms over these data. We can represent a memory location, containing an integer i, simply as a cell holding the value. We can represent a list of cells with the addition of a nil node.

In the diagram above, the m node has one principal port that will be used to interact with it, and one auxiliary port to connect to the remaining elements of the list. The nil node just has one principal port, and no auxiliary ports. To simplify the diagrams, we often just write the contents of the node and omit the name when no confusion will arise. For example, here is a list of 4 elements:



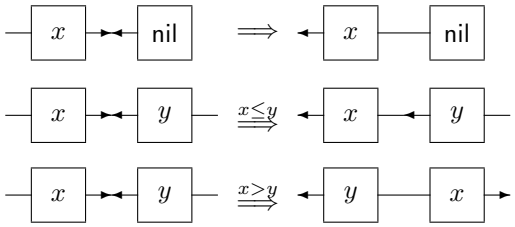
We remark that this diagrammatic representation of the dynamic list data structure is not the same as what one would usually draw, as the arrows are not pointers, but they represent the principal ports which dictates how we can interact with the data structure.

3 Examples: Sorting

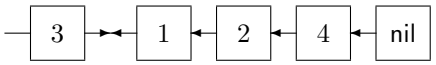
Suppose that we wanted to insert an element into a sorted list. In Java we might write the code given below

```
static List insert(int x, List l) {
  if (isEmpty(l) || x <= l.head) return cons(x, l);
  else {
    l.tail = insert(x, l.tail);
    return l;
  }
}
```

When teaching dynamic data structures, we might begin by drawing a diagram for this algorithm, which would consist of the list before the insertion, then a number of modifications of the diagram, yielding the final diagram. To derive the code above from such a diagram is not an automatic process, and is of course subject to error. In interaction nets we would simply write the program directly as the rules needed to transform the data structure:



These three rules explain all the computation. We leave the reduction of the following net as an exercise for the interested reader, which will rewrite to the list of 4 elements given above using three applications of the above rules. The implementation of insertion sort is a straightforward extension to this example.

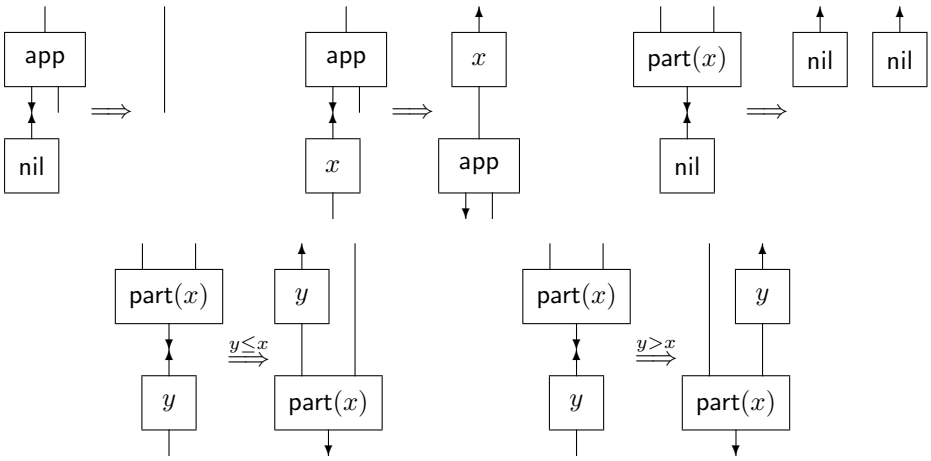


If we compare the interaction rules with the Java program we find that we can follow them step-by-step. However, the three diagrams contain all the implementation details—there are no notions of procedures (methods), conditions, recursion, etc. as syntax, as they are all absorbed into the rules. The relationship between Java programs and interaction nets is not the purpose of this paper; in general we will find that the Java version will contain very different control information than the interaction nets version. However, it is worth remarking that the interaction net program can be seen as directly manipulating the internal data structure, which is one of the main features of this approach.

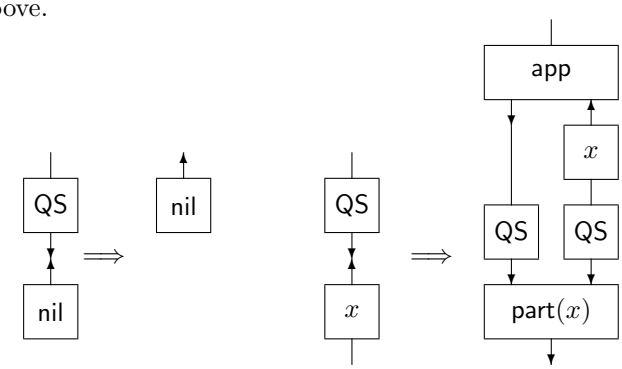
Our next example to demonstrate how easily one can program with interaction nets also brings out the relationship between the diagrammatic representation of the algorithm and the intuition of the algorithm. We give an implementation of the Quicksort algorithm, using again the linked list structure. We need some auxiliary rules to concatenate two lists, and also to partition a list into two lists. We begin with concatenation, which can be expressed in Java as follows:

```
static List concat(List l1, List l2) {
  if (isEmpty(l1)) return l2;
  else {
    List l = l1;
    while (isCons(l.tail)) l = l.tail;
    l.tail = l2;
    return l1;
  }
}
```

The algorithm performed in both interaction nets and the Java version is essentially the same: traverse the first list and connect its last element to the start of the second list. This is given by the first two rules below. We also give the three rules for the partition of a list, which is the core of the Quicksort algorithm, splitting a list into two lists.



We can now put together the main algorithm for Quicksort which is given by the following two rules. QS is the node to represent Quicksort, and we use app and part from above.



It is worth pointing out some salient features of this implementation of the Quicksort algorithm:

1. The graphical representation of the problem is directly cast into the graphical language. The algorithm given can be understood as programming directly with the internal data structures, rather than some syntax describing it.
2. All rewrite steps correspond to steps in the computation: there is no need to introduce additional data structures and operations that are not part of the problem.
3. Because of point 1 above, if we single-step the computation we get an animation of the algorithm directly from the rewriting system.

4 Discussion

When learning data structures in programming languages, specifically dynamic structures with pointers, we find that the diagram used to explain the problem and the code are very different. In particular, the diagrams do not always show the temporal constraints, and therefore converting the diagrammatic intuitions into programs can be quite difficult. With interaction nets we draw the diagrams once, and this gives the program directly. Because of the confluence result, it does not matter which order we apply the rules, thus eliminating the temporal constraints. It is the programmer that draws the diagrams, and the diagrams explain all of the computation (nothing is replaced by code, like while loops, etc.). The diagrams are then implemented directly (and moreover, they are data structures that are well adapted to implementation).

There are a number of textual programming languages for interaction nets in the literature (see for instance [8, 3, 10]), and one of the future developments needs to be tools to offer a visual representation of these. Being able to convert the textual representation to diagrams would be a useful tool for debugging, but this does not offer the ability to directly manipulate nets and rules in a uniform way. Some developments are currently underway to address these issues, and preliminary experiments show that this direction is an exciting way to write programs like the ones described in this paper. Nevertheless, this is a topic of current research very much in its infancy.

What we have presented leads to a very direct form of visual programming. We can imagine tools to display animations of traces of executions which will follow the intuitions of the programmer (each step is one rule). Because interaction nets explain all the elements of the computation it is also a useful debugging tool as well as an educational tool for data structures.

Interaction nets have been used for representing sharing in computation. This is obviously still valid in the graphical approach. Having a language which is at the forefront or research into optimal computations is clearly an advantage. As we have hinted above, they are also amenable to parallel computations: the one step confluence property of rewriting together with the fact that all rules are local means that we can apply all redexes at the same time [11]. Interaction nets may therefore have potential to represent parallel algorithms visually.

The next developments that are needed are tools to offer direct manipulation of interaction nets to investigate the potential as a programming language and study usability issues. This relies on the development of editing tools and techniques to write modular programs, hierarchical structures, etc. The development of such an environment will be the subject of a future paper.

5 Conclusion

Interaction nets are a graphical (visual) programming language used successfully in other branches of computer science. Our aim in this paper was to demonstrate by example that they are well adapted to visual programming, with the main goal

to introduce this formalism to the *diagrams* community through examples, and hint as some of the possible potential of this formalism that is currently under development. We have achieved this by extending the interaction net formalism, and given some new example programs to exhibit these extensions.

Acknowledgements. This work was partially supported by CRUP, Acção Integrada Luso-Britânica N.B-40/08, and the British Council Treaty of Windsor Programme.

References

1. L. Dami and D. Vallet. Higher-order functional composition in visual form. Technical report, 1996.
2. M. Fernández and I. Mackie. Interaction nets and term rewriting systems. *Theoretical Computer Science*, 190(1):3–39, January 1998.
3. M. Fernández and I. Mackie. A calculus for interaction nets. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 170–187. Springer-Verlag, September 1999.
4. M. Fernández and I. Mackie. Operational equivalence for interaction nets. *Theoretical Computer Science*, 297(1–3):157–181, February 2003.
5. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, Jan. 1992.
6. K. Hanna. Interactive Visual Functional Programming. In S. P. Jones, editor, *Proc. Intnl Conf. on Functional Programming*, pages 100–112. ACM, October 2002.
7. J. Kelso. *A Visual Programming Environment for Functional Languages.* PhD thesis, Murdoch University, 2002.
8. Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, Jan. 1990.
9. I. Mackie. Efficient λ-evaluation with interaction nets. In V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 155–169. Springer-Verlag, June 2004.
10. I. Mackie. Towards a programming language for interaction nets. *Electronic Notes in Theoretical Computer Science*, 127(5):133–151, May 2005.
11. J. S. Pinto. *Parallel Implementation with Linear Logic.* PhD thesis, École Polytechnique, February 2001.
12. H. J. Reekie. *Realtime Signal Processing – Dataflow, Visual, and Functional Programming.* PhD thesis, University of Technology at Sydney, 1995.