Electronic Communications of the EASST
Volume X (2007)

Proceedings of the
Sixth International Workshop on
Graph Transformation and Visual Modeling Techniques
(GT-VMT 2007)

Visual Programming with Recursion Patterns in Interaction Nets

Ian Mackie, Jorge Sousa Pinto and Miguel Vilaça

12 pages

Guest Editors: Karsten Ehrig, Holger Giese
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Visual Programming with Recursion Patterns in Interaction Nets

Ian Mackie[1], **Jorge Sousa Pinto**[2] **and Miguel Vilaça**[2]

[1] LIX, École Polytechnique, 91128 Palaiseau Cedex, France
[2] Departamento de Informática, Universidade do Minho , 4710 Braga, Portugal

Abstract: In this paper we propose to use Interaction Nets as a formalism for Visual Functional Programming. We consider the use of recursion patterns as a programming idiom, and introduce a suitable archetype/instantiation mechanism for interaction agents, which allows one to define agents whose behaviour is based on recursion patterns.

Keywords: Interaction nets

1 Introduction

This paper is about visual programming with Interaction Nets, a graph-rewriting formalism introduced by Lafont [9], inspired by Proof-nets for Multiplicative Linear Logic. In Interaction Nets, a program consists of a number of interaction rules and an initial net that will be reduced by repeated application of the rules. The formalism combines the usual advantages of visual programming languages, but with the following additional features:

– Programs and data structures are represented in the same framework, which is useful for tracing and debugging purposes;
– All aspects of computations, such as duplication and garbage-collection, are explicit.

Interaction Nets have been extensively used for functional programming as an efficient intermediate (or implementation) language. In particular, functional programs can be *encoded* in Interaction Nets, using one of the many encodings of the λ-calculus. Section 3 reviews how a functional language can be encoded in Interaction Nets following this approach, without entering the details of any particular encoding of β-reduction. The focus of this paper will be the adequate treatment of inductive datatypes, pattern-matching, and recursive function definitions.

The remaining sections of the paper introduce and systematise the use of a functional style for programming with Interaction Nets with *recursion patterns*, and introduce a new construct (the *archetype*, Section 4) which captures the behaviour of recursion patterns. We claim that this style is a good choice for defining and executing visual functional programs.

The style of programming we refer to is widely used by functional programmers: it is based on programs that perform iteration on their arguments, usually known in the field as *folds*, and (the dual notion) programs that construct results by co-iteration, *unfolds*. Among other advantages of using folds and unfolds for programming, they can be composed to construct complex recursive programs, and they are particularly adequate for equational reasoning: proofs of equality can be done using a *fusion law* instead of recursion.

The body of theoretical work on recursion patterns comes from the field of *datatype-generic programming* (see [4] for an introduction), which studies these patterns in a datatype-parameterized way. The examples in the paper use lists, but it is straightforward to generalize the ideas to arbitrary regular datatypes.

Section 5 introduces interaction net programming with recursion patterns; Section 6 and Section 7 then present archetypes for folds and unfolds respectively. Section 8 concludes and discusses future work.

2 Background

Recursion Patterns. The ideas developed in this paper for Interaction Nets are very much inspired by Functional Programming. One fundamental aspect that we will use extensively is the ability to use a set of *recursion patterns* for each datatype. For instance few Haskell programmers would write a list sum program with explicit recursion as

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

Most would define `sum = foldr (+) 0`, where `foldr` is a recursion pattern encoded as the following higher-order function:

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

A function like `sum` is often called a *fold*. The use of recursion patterns has the advantage of being appropriate for program transformation and reasoning using the so-called *calculation-based* style [1]. To see how less obvious folds can be defined, consider the list *append* function:

```
app :: [a] -> [a] -> [a]
app     []   l = l
app (x:xs)   l = x:(app xs l)
```

This is a higher-order fold that iterates over its first argument to produce a function (`id` is the identity function): `app = foldr (\x r l -> x:(r l)) id`.

The dual notion of fold is the co-recursive *unfold* that allows one to produce lists whose tails are constructed recursively by the function being defined. For instance the Haskell function

```
downfrom 0     = []
downfrom (n+1) = (n+1):(downfrom n)
```

can be written alternatively as `downfrom = unfold (==0) id pred` where `pred` is the predecessor function, and `unfold` is defined as follows

```
unfold :: (t -> Bool) -> (t -> a) -> (t -> t) -> t -> [a]
unfold p f g x = if p x then [] else f x : unfold p f g (g x)
```

One of the reasons why unfolds are important [5] is that together with folds they give us back the power of arbitrary recursion: the composition of a fold with an unfold is a function (known as a *hylomorphism* [14]) whose recursion tree is the intermediate structure constructed by the unfold. In a language with a sufficiently rich type system, most useful recursive functions can be defined in this way.

Interaction Nets. An interaction net system [9] is specified by giving a set Σ of symbols, and a set $\mathscr{R}$ of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occurrence of a symbol $\alpha \in \Sigma$ will be called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a distinguished one called the *principal port*, and n *auxiliary ports* labelled $x_1, \ldots, x_n$.

A net built on Σ is a graph (not necessarily connected) where the nodes are agents. The edges between nodes of the graph are connected to *ports* in the agents, such that there is at most one edge connected to every port in the net. Edges may be connected to two ports of the same agent. Principal ports of agents are depicted by an arrow. The ports of agents that have no edge connected are called the *free ports* of the net. The set of free ports define the *interface* of the net.

The dynamics of Interaction Nets are based on the notion of *active pair*: any pair of agents (α, β) in a net, with an edge connecting together their principal ports. An *interaction rule* $((\alpha, \beta) \Longrightarrow N) \in \mathscr{R}$ replaces an occurrence of the active pair (α, β) by the net N. Rules must satisfy two conditions: the interfaces of the left-hand side and right-hand side are equal (this implies that the free ports are preserved during reduction), and there is at most one rule for each pair of agents, so there is no ambiguity regarding which rule to apply.

If a net does not contain any active pairs then we say that it is in normal form. We use the notation $\Longrightarrow$ for one-step reduction and $\Longrightarrow^*$ for its transitive reflexive closure. Additionally, we write $N \Downarrow N'$ if there is a sequence of interaction steps $N \Longrightarrow^* N'$, such that N' is a net in normal form. The strong constraints on the definition of interaction rules imply that reduction is strongly commutative (the one-step diamond property holds), and thus confluence is easily obtained. Consequently, any normalizing interaction net is strongly normalizing.

As an example, we show a system for representing lists of numbers. Lists are inductively defined by an agent Nil of arity 0 representing the empty list, and an agent Cons of arity 2 representing a cell in the list, containing an element and a tail list. Lists are constructed such that the principal port of each Cons agent corresponds to the root of the list.

To implement, for instance, list concatenation, we need an additional binary agent app. Concatenation is defined recursively on one of the argument lists, as expected. As such, the principal port of the agent must be used for interacting with this argument. The necessary interaction rules are given in Figure 1, together with an example net, representing the concatenation of lists [1,2] and [3,4].

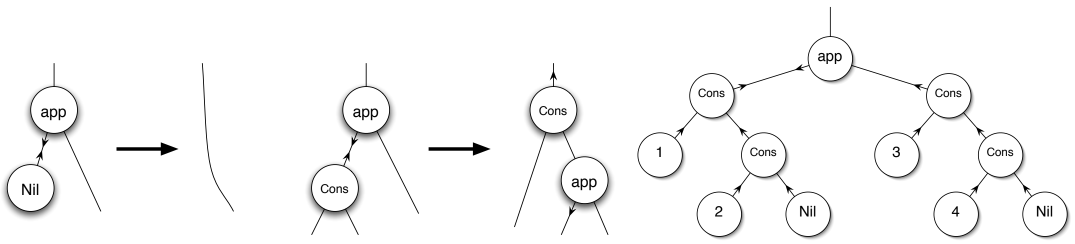


Figure 1: Interaction rules of agent app and an example net

Thus, an implementation of list concatenation can be obtained by Σ containing $\{\mathsf{Nil}, \mathsf{Cons}, \mathsf{app}\}$, with arity 0, 2, 2 respectively, and $\mathscr{R}$ consisting of the rules in Figure 1.

Related Work: Visual Functional Programming. Work in this area has addressed different aspects of visual programming. The Pivotal project [6] offers a visual notation (and Haskell programming environment) for data structures, not programs. Reekie's Visual Haskell [15] 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 one to define functional programs visually and then reduce them step by step. Finally, VisualLambda [2] 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.

Visual Haskell and VisualLambda have in common that functions are represented as boxes with input ports for the arguments and an output port for the result; the contents of the box correspond to the body of the function. They differ in that Visual Haskell uses variables to refer to function arguments, while VisualLambda uses a purely graphical notation based on arrows.

Kelso's VFP uses a notation without boxes, more inspired by the traditional representations of functional programs used in implementation-oriented abstract machines (see Section 5). In particular, it allows for named functions but also for λ-abstractions, and an explicit application node exists. Variables are used for arguments, as in Visual Haskell.

Higher-order programming is a fundamental feature of functional programming. A function f can take function g as an argument and g can then applied within the body of f. Expressing this feature is easy if variables are used as in Visual Haskell and VFP; in VisualLambda a special box would be used as a placeholder for g (in the body of f) to be instantiated later, and an arrow would link an input port in the box of f to the box of g.

The work presented in the present paper uses a pure visual representation of programs, without variables. In this aspect it resembles VisualLambda, however our work differs significantly from this in that no boxes are used, and all the graph-rewriting operations are *local* in the sense that only two nodes of the graph are involved in each step.

A second difficulty arising from the higher-order nature of programs is that a (curried) function of two arguments may receive only its first argument and return as result a function. In a box-based representation this means that it must be possible for a box to lose its input ports one by one—a complicated process. Interaction nets treat this problem naturally as will become clear. Moreover in this paper we introduce a new notion (the archetype), which captures precisely the behaviour of many typical curried functions.

3 Visual Functional Programming with Interaction Nets Using Explicit Abstraction and Application Nodes

In this section we explain how a very simple functional programming language can be *encoded* in Interaction Nets. The language has inductive types, pattern-matching on these types, and explicit recursion.

We first review the basic principle shared by most well-known encodings [13, 12, 16] of the λ-calculus into Interaction Nets, and show how this basic language can easily be extended to cover other features of functional languages.

The usual way of representing functional programs with interaction nets is based on a pair of symbols λ, @ of arity 2, such that a β-reduction step corresponds to an interaction between an agent λ and an agent @. These representations are based on an explicit depiction of the λ-abstractions in the program, as well as applications of functions to arguments.

While this may be visually more complicated than the boxes used by some of the systems reviewed in Section 2, it certainly solves the "higher-order" problem in a natural way, since function arguments are treated like any other arguments. The definition of the application function `ap f x = f x` illustrates this point (see Figure 2 left).

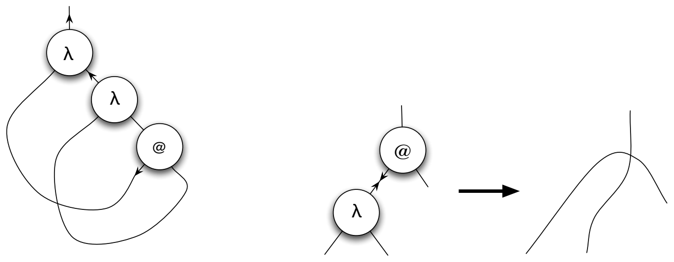


Figure 2: `App` definition (left) and β rule (right)

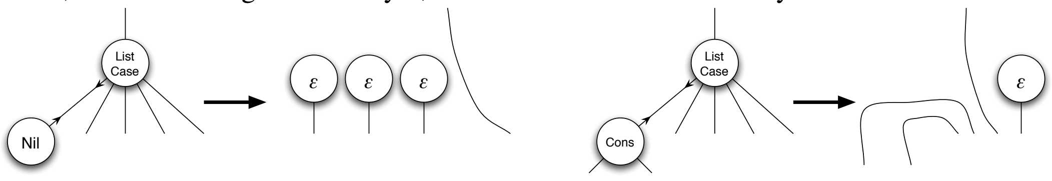
We only discuss the features of the linear λ-calculus here, which is shared by all encodings. It is beyond the scope of this paper to unclude the details of the non-linear aspects, but refer the reader to [12, 13] for the encodings of the full λ-calculus.

Consider an interaction system containing the symbols $\{@, \lambda\}$ as explained above, as well as the β interaction rule of Figure 2 (right).

This system defines the visual programming language for the linear λ-calculus. A visual functional program consists of this interaction system, together with a closed functional expression to be evaluated, represented by a net with a single free port. We now outline how other features can be introduced in the interaction system to enrich this core language.

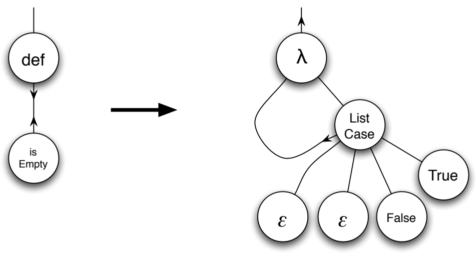
The next feature is Inductive Types and Pattern-matching. Consider a datatype T with constructors $C_1 \ldots C_n$, with arities $a_1 \ldots a_n$. This can be modelled in a straightforward way by an interaction system containing n agents labelled C_i with arity a_i, $i = 1 \ldots n$; values of type T correspond to closed trees built exclusively from these agents (in a tree all principal ports are oriented in the same direction). In a constructor agent, auxiliary ports are input ports, and the principal port is an output port. An example of this is the datatype of lists with constructors Nil and Cons, as in Figure 1.

Pattern-matching over an inductive type T can be implemented by a special agent Tcase. For instance, the ListCase agent has arity 5, and its behaviour is defined by the two rules:

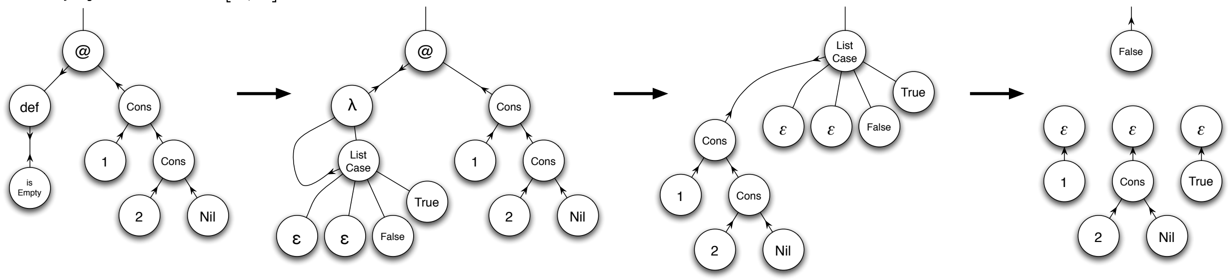


Here two different nets are connected to the ListCase agent. One is a net to be returned when the argument list is empty, and the other is a net with three ports, used to combine the head and tail of a non-empty list. Observe that one of these nets is not used in each rule, and must be erased with ε agents. The interaction of an ε agent with any other agent erases the latter and propagates new ε agents along its auxiliary ports.

The approach outlined above allows for unnamed functions only. In a visual language one would like to have the possibility of defining named functions, which would most naturally correspond to agents in the interaction system. A special agent def can be used for unfolding named function definitions. For instance a function isEmpty can be defined by the following interaction rule for the agent def:



The following figureshows the reduction of the visual program corresponding to the application of isEmpty to the list $[1,2]$.



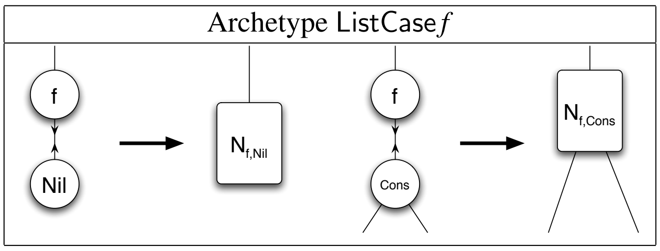
This agent also allows for a visually appealing treatment of recursion: it suffices that the right-hand side of interaction rules involving def reintroduces an active pair (the left-hand side of the rule) as a sub-net.

To sum up, Interaction Nets allow to visually represent functional programs and data structures in the same very simple formalism; moreover higher-order features, which are a typical difficulty in the visual setting, are treated in a natural way, and the execution of the program can be efficiently implemented within the formalism.

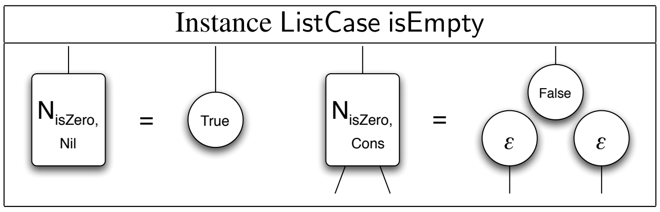
4 Agent Archetypes

Although no standard programming language exists as a front-end for programming with Interaction Nets, it is generally well accepted that any such language should contain some form of support for modularity and reusability. In particular, a mechanism should exist to facilitate the definition of interaction rules that follow identical patterns.

To illustrate, consider again the app agent of Figure 1. It is defined by case analysis on the structure of the argument, and in fact any other agent defined in this way must have two interaction rules with a similar structure to those in Figure 1. We now introduce a concept designed precisely to isolate this structure, which we designate *archetype*. The ListCase archetype given below should be interpreted as follows: any agent f that fits this archetype interacts with both Nil and Cons, and the right hand sides of the corresponding rules are nets to be instantiated, that will be called respectively $N_{f,\mathsf{Nil}}$ and $N_{f,\mathsf{Cons}}$.



To define a new agent following the archetype, an *instance* is created, by simply providing the nets in the right-hand side of the interaction rules. This implicitly includes the agent declaration, as well as the instantiated interaction rules for this agent, in the interaction system being defined. As an example, the isEmpty agent (and its behaviour) is defined as an instance of the ListCase archetype. ε agents are used to explicitly *erase* the head and tail of the list, which are not used in the result.



For a second example, take the agent def used in Section 5 for function definitions. We create an archetype for *defined functions*, whose only mandatory rule is for interaction with def, with the right-hand side to be instantiated.

Recursive archetypes are most interesting, and will be very useful in the rest of the paper (in Section 6 a recursive archetype will be given for the app agent). Although archetypes can be useful for programming with interaction nets in general, our examples of using them here concern features of functional programming languages.

The ListCase example shows that archetypes allow for a natural treatment of higher-order concepts: ListCase can be seen as a function that takes certain arguments (the instantiated nets) and returns another function (an agent with its own rules) as result.

5 Interaction Net Programming with Recursion Patterns

Section 3 was about using Interaction Nets for *encoding* functional programs, with the practical goal of producing efficient functional compilers. From the point of view of visual programming, the drawbacks of this approach are that the introduction of explicit abstraction and application nodes complicates the visual representation of programs (the same is true of explicit case constructs), and also raises the matching duplicators problem. Solving this problem implies introducing in the system machinery that destroys the clean visual representation of terms.

Our goal in this paper is to propose a number of principles and extensions for direct visual programming with Interaction Nets, in a *functional style*. To see what we mean by *direct*, consider again the interaction rules given in Figure 1 . It is easy to see that both define a behaviour for the agent app similar to the standard list concatenation function, which can be written in Haskell as shown in Section 2.

In both cases, a program consists of a collection of function definitions encoded directly as interaction rules in a particular interaction system, together with a closed functional expression to be evaluated in the context of those definitions, represented by a net with a single free port. While in the second approach a function definition corresponds to interaction rules for a special agent def, in Figure 1 there is a direct correspondence between the clauses in the definition of a function f and the interaction rules defining the behaviour of the agent f. A comparison of both definitions reveals that the first approach is visually simpler, and thus more appropriate for representing programs, than the second, standard approach.

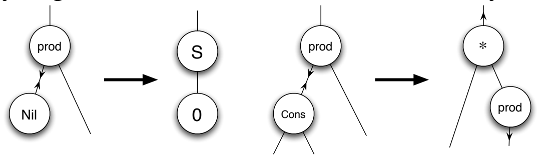
Naturally, there are important limitations to the class of programs with which the simplified representation can be used. The example above takes advantage of a fundamental aspect of Interaction Nets, which is that pattern-matching on the outermost constructor is *built-in* through the rule selection mechanism (in the definition of app in Figure 1, only the outermost constructor is matched). Matching deeper constructors, or matching more than one argument in the same clause, would force us to use an explicit case agent.

For a certain class of patterns (match-sequential systems [17]) there are known transformations which result in a system which examines arguments one at a time. We can use this transformation to obtain an explicit system. We refer the reader to [8] for a detailed presentation of one such transformation.

It will now be shown that the use of a programming style based on recursion patterns allows precisely for the direct representation to be used, since these operators perform pattern-matching on a single top-level constructor.

Iteration: Fold Agents. The simplest form of recursion is iteration, which substitutes the datatype constructors by some given functions. Taking lists as an example, a fold is a function that combines the head of the list with the result of recursively applying the function to the tail of the list, to produce the result (a given value is returned at the end of the list).

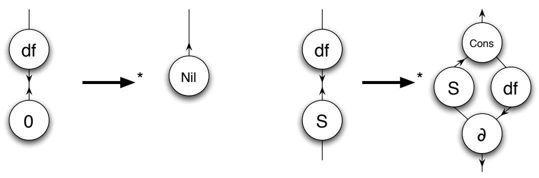
Since this requires matching on the outermost constructor only, iteration neatly fits the interaction paradigm. The following is the definition of the *product* fold, which computes the product of all the numbers in a list (it uses an agent $*$ for multiplication of numbers; we assume the arithmetics to be correctly implemented in the current interaction system).



It is easy to see that other, more powerful recursion patterns on an inductive type can be captured in the same way. For instance, *primitive recursion* on a list would allow for the *tail* of the list itself to be used as well.

Co-recursion Patterns: Unfold Agents. *Co-recursive* functions provide structured ways to construct values of recursive types. Taking lists as an example again, the *unfold* co-recursion pattern, which is the dual of fold, corresponds to functions that construct lists by giving an element to be placed at the head position, together with a seed used as an argument to recursively construct the tail of the list. This is the simplest form of co-recursion.

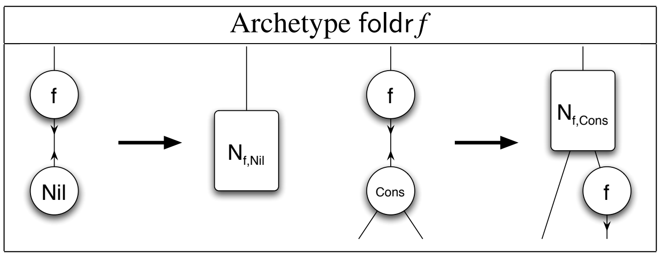
Let us consider an unfold agent *downfrom* (df) which interacts with a natural number n to construct the list containing all the numbers from n down to 1, in this order. Its rules are the following



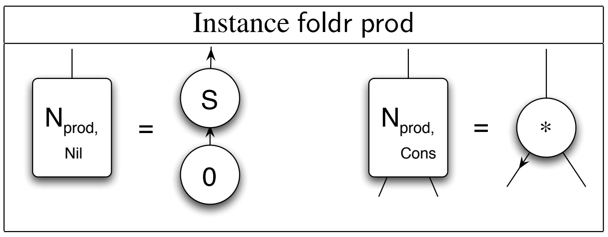
6 Fold Archetypes

The rules that characterize the behaviour of a fold agent (on a particular inductive type) can be described by a *recursive* archetype, in the sense that the parameterized agent occurs in the right-hand side of one of the rules.

Taking the case of lists, interaction rules must be defined for f to interact with both Nil and Cons. The archetype is



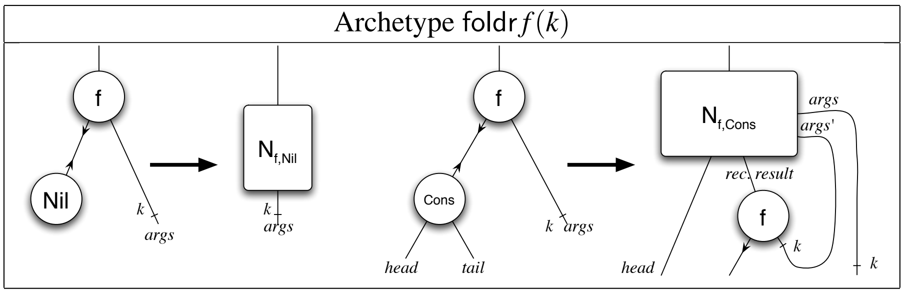
where interaction with Nil results in an arbitrary net, and interaction with Cons sends an f agent along the tail of the argument list, and a net $N_{f,\mathsf{Cons}}$ then combines the head of the list with the recursive result. As an example, the agent prod can be alternatively defined as the following instance.



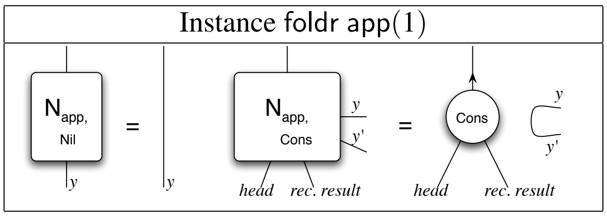
The principles developed above can be applied to folds over any regular inductive type.

Higher-order Folds. Our current definition of a fold agent is still not satisfactory, and will now be generalized. Consider again the list *append* function. As seen in Section 2, this is a higher-order function of two arguments, defined by recursion on its first argument. This fold can be defined with Interaction Nets as a binary agent (see Figure 1), which clearly does not match our current definition of the fold archetype.

Functions of more than one argument defined as folds over one of the arguments lead us to the generalization of the foldr archetype, as shown in the following figure . This is parameterized by the number of extra arguments of the fold agent; our previous definition is of course a particular case of this where $k = 0$.

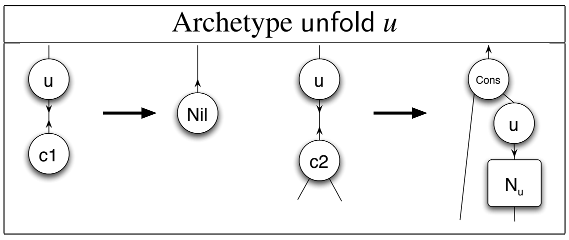


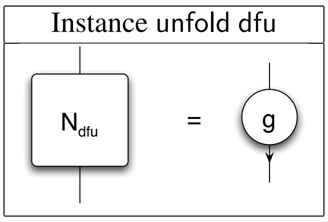
The definition of *append* as an instance of this archetype, for $k = 1$, can be seen below. The open wire in the net $N_{\mathsf{app,Cons}}$ corresponds to the fact that the second argument of the fold is preserved in the recursive call.



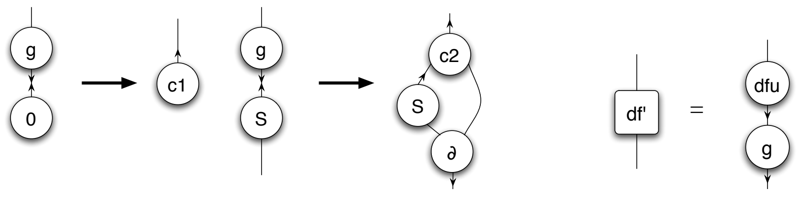
7 Unfold Archetypes

It is less obvious to define an archetype for unfolds. Consider again the case of lists; it is clear that there must be two rules (to produce empty and non-empty lists respectively); the contents of the right-hand side of each interaction rule is also clear (with a parameter net appearing in the second rule). However, since the arguments of an unfold are arbitrary, the two rules cannot correspond to interactions between the unfold and its arguments. Instead, we will consider two special agents that interact with the unfold.





Now observe that an instance of this archetype does not immediately behave as an unfold; it must be connected to the net $\mathsf{N_u}$. To take *downfrom* as an example again, this net is in this case simply an agent (see right of previous figure) whose behaviour is given by the rules on the left in the following. Then the following macro (on the right) can be defined:



8 Conclusions and Future Work

One of our long-term goals is to develop a full environment for interaction net programming—a tool is currently being developed. We are currently working on the archetype definition and instantiation mechanismSubsequently, we plan to incorporate in the programming environment a mechanism that generates the appropriate archetype for a given user-defined inductive type.

This paper opens a number of other research questions at the theoretical level. One of the main reasons for using recursion patterns and datatype-generic programming is that this style of programming is good for *reasoning* about programs equationally. The work in this paper allows us to reason about functional programs *visually*. In [11] we derive a *fusion law* for the fold archetype, that already makes it possible to transpose to the visual setting classic program transformation techniques such as the introduction of accumulators or tupling.

A different approach that we intend to explore is to establish a formal correspondence between a core functional language with recursion patterns and an interaction system for that language. This will allows us to be more precise in the study of the calculation laws for visual programs.

At the level of the programming environment, the graphical notation then becomes an alternative to writing functional programs textually. The environment should be able to translate between visual programs and textual programs, and all operations performed on programs at the visual level correspond closely to the same operations at the expression level.

Acknowledgements: The work of Vilaça was granted by FCT (SFRH/BD/18874/2004).

Bibliography

[1] R. Bird. The Promotion and Accumulation Strategies in Transformational Programming, *ACM Transactions on Programming Languages and Systems*, **6**(4), October 1984, 487–504.

[2] L. Dami and D. Vallet. Higher-order functional composition in visual form. Technical report, University of Geneva, 1996.

[3] M. Fernández. Type assignment and termination of interaction nets. *Mathematical Structures in Computer Science*, 8(6):593–636, 1998.

[4] J. Gibbons. Calculating Functional Programs. In *Proceedings of ISRG/SERG Research Colloquium*. School of Computing and Mathematical Sciences, Oxford Brookes University, 1997.

[5] J. Gibbons and G. Jones. The Under-appreciated Unfold. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 273–279. ACM Press, 1998.

[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] J.R. Kennaway. Implementing term rewrite languages in DACTL. *Theoretical Computer Science*, 72:225–249, 1990.

[9] 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.

[10] M. Fernández and I. Mackie. Coinductive techniques for operational equivalence of interaction nets. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS'98)*, pages 321–332. IEEE Computer Society Press, June 1998.

[11] I. Mackie, J. S. Pinto, and M. Vilaça. Functional programming and program transformation with interaction nets. Technical Report DI-PURe-05.05.02, Universidade do Minho, 2005.

[12] I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, September 1998.

[13] I. Mackie. Efficient λ-evaluation with interaction nets. In V. van Oostrom, editor, *Proceedings of Rewriting Techniques and Applications: 15th International Conference (RTA 2004)*, volume 3091 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[14] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.

[15] H. J. Reekie. *Realtime Signal Processing – Dataflow, Visual, and Functional Programming*. PhD thesis, University of Technology at Sydney, 1995.

[16] F.-R. Sinot. Token-passing Nets: Call-by-need for Free. In *Proceedings of the 1st. International Workshop on Developments in Computational Models (DCM'05)*, 2005.

[17] S. Thatte. A refinement of strong sequentiality for term rewriting systems with constructors. *Information and Computation*, 72:46–65, 1987.