Functional Programming and Program Transformation with Interaction Nets

Ian Mackie, Jorge Sousa Pinto, and Miguel Vilaça

`ian@dcs.kcl.ac.uk, jsp@di.uminho.pt, jmvilaca@di.uminho.pt`

Functional Programming and Program Transformation with Interaction Nets
by Ian Mackie, Jorge Sousa Pinto, and Miguel Vilaça

Abstract

In this paper we propose to use Interaction Nets as a formalism for Visual Functional Programming. We consider the use of recursion patterns and introduce a suitable archetype/instantiation mechanism for interaction agents. We also consider program transformation by *fusion*, a well-known transformation technique, and show that this extends smoothly to our visual programming framework. Examples of applying this technique include transformations of two-pass functions into single-pass ones, and the introduction of accumulations.

1   Introduction

Many attempts [3, 4, 7] have been advanced to create a visual notation for functional programs (or even a full-fledged visual functional programming language), but as far as we know none of these has been successful or widely used in practice.

In this paper we propose to use an existing formalism for the visual representation of functional programs. *Interaction Nets* (INs) are a *graph-rewriting* formalism introduced by Lafont [5], inspired by Proof-nets for Multiplicative Linear Logic. From a programming point of view, interaction nets can be seen as a visual programming language in themselves; however they have been put to use more fruitfully as an implementation language, to encode programs (of a core functional language) in clever ways that allow for the close control of shared reductions and evaluation strategies.

Our interest here is on a totally different connection between INs and functional programming: we are interested in using nets as a tool for visual functional programming. We show how INs can be used:

– to simply *represent* functional programs visually and to *animate* the execution of such programs by graph-rewriting;
– to *reason* about functional programs and to perform *visual program transformation*, using the standard *fusion* technique of Functional Programming, here transposed to the visual setting.

It is important to stress that although many standard examples of INs are functional by nature, which is the aspect developed in this paper, many other applications involve nets that do not have a functional interpretation.

Structure of the Paper. Section 2 reviews the basic concepts of Interaction Nets; in section 3 some principles are informally introduced showing how visual functional programs can be defined with INs. Section 4 introduces a new mechanism for programming with INs, that we call *archetypes*, and section 5 reviews the use of *folds* for functional programming, as well as program transformation by *fusion*. It is shown how an archetype can be used to capture the behaviour of a fold agent over a particular regular data type. In section 6 we see that this is inadequate for capturing the definition of higher-order folds. A suitable generalization of fold archetypes is then given. Section 7 presents a visual fusion law for folds over lists, together with many examples of its application to visual program transformation. Section 8 concludes the paper and discusses future work.

2   Interaction Nets

An interaction net system [5] is specified by giving a set Σ of symbols, and a set $\mathcal{R}$ 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* depicted

by an arrow, and *n auxiliary ports* labeled $x_1, \ldots, x_n$ corresponding to the arity of the symbol.

A net built on Σ is a graph (not necessarily connected) with agents at the vertices. The edges of the graph connect agents together at the ports such that there is only one edge at every port (edges may connect two ports of the same agent). The ports of an agent that are not connected to another agent are called the free ports of the net, and define its *interface*. There are two special instances of a net: a wiring (no agents), and the empty net.

A pair of agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected together on their principal ports is called an *active pair*; the interaction net analog of a redex. An interaction rule $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ replaces an occurrence of the active pair (α, β) by a net N. Rules must satisfy two conditions: the interfaces of the left-hand side and right-hand side are equal (this implies that all the free ports are preserved during reduction), and there is at most one rule for each pair of agents.

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 an interaction rule 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 a very simple example of an interaction system, an implementation of list concatenation can be obtained by Σ containing $\{\mathsf{Nil}, \mathsf{Cons}, \mathsf{app}\}$, with arity 0, 2, 2 respectively, and $\mathcal{R}$ consisting of the rules in figure 1. Naturally, this presupposes that the system contains some base elements from which to build lists.
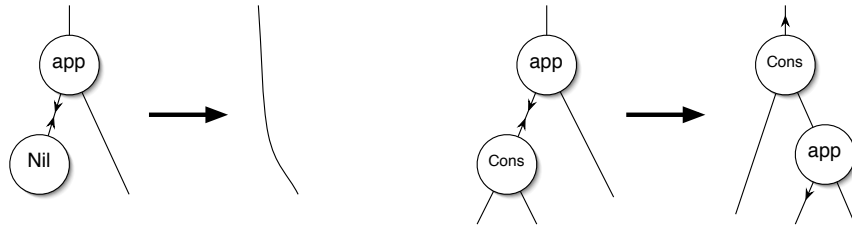

Fig. 1. IN definition of *append*

Typing. The type discipline usually considered for INs uses a set of constant types. Every port is assigned a type positively (for *output* ports) or negatively (for *input* ports). In a well-typed net every edge connects a positive and a negative occurrence of the same type.

Equivalence of Interaction Nets and Weak Reduction. A notion of *canonical form* is defined for Interaction Nets; a particular case is a net whose interface contains principal ports only. An adequate notion of evaluation consists in doing the minimum amount of work to reach a canonical form (rather than fully reducing nets); in particular, this will never reduce active pairs inside *disconnected* components of a net, which roughly corresponds to lazy evaluation in functional programming.

The adequate notion of equivalence for INs is *contextual equivalence*: two nets are equivalent if their canonical forms have the same interface, whatever nets are connected to them. This is the same as saying that the nets are indistinguishable in any context. Because reduction does not necessarily terminate, the adequate technique for proving equivalence is *bisimilarity*, a method based on comparing the transition trees containing all possible sequences of observations of both nets (this is a coinductive method since a notion of *greatest bisimulation* is used). See [6] for full details. The following is an example of contextual equivalence:
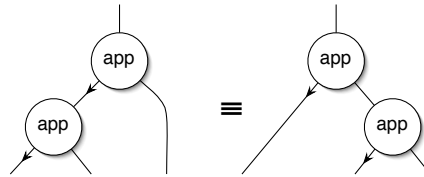
3 Functional Programming with Interaction Nets

Representing Inductive Types. Consider a datatype T with n constructors $C_1 \ldots C_n$, with arities $a_1 \ldots a_n$. This can be modeled in a straightforward way by an interaction system containing n agents labeled 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). A function over such a type can then be encoded as an agent with appropriate interaction rules for the constructors of the type. In a constructor agent, auxiliary ports are input ports, and the principal port is an output port.

An example of this is given by the datatype of lists with constructors Nil and Cons, as in the example of figure 1. The following is an example of a value of type List Int, where we consider that integers are represented by an infinite family of agents (alternatively this can be seen as a single agent carrying a value of a basic type built into the system).
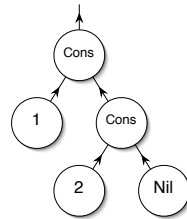
Pattern-matching, Function Definitions, and Recursion. A fundamental aspect of interaction nets is that pattern-matching is *built-in* through the rule selection mechanism. Consider the Haskell implementation of a list concatenation function

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

It is easy to see that the interaction rules given in figure 1 define a behaviour for the agent `app` similar to this, where each interaction rule corresponds to a clause of the function definition; the appropriate clause is selected by matching on the first argument, which corresponds to the principal port of the agent. Although we omit typing considerations here, it is immediate to see that input ports (negatively typed) correspond to function arguments, and the unique output port (positively typed) corresponds to the result.

A visual functional program then consists of a net (containing a single free port, corresponding to a *closed* functional expression) to be reduced in the context of an interaction system defining a particular set of functions. For instance the following net corresponds to the expression `app [1,2] [3,4]` to be evaluated with the interaction system of figure 1.
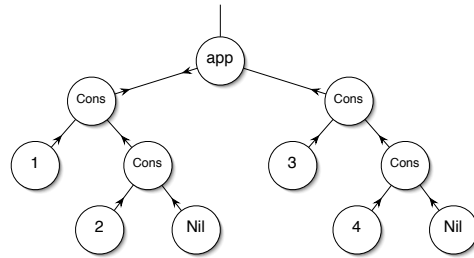
Higher-order Programming. Naturally, the above approach is only possible because in the definition of `app` only the outermost constructor is matched. Matching deeper constructors, or matching more than one constructor simultaneously, would be possible in Haskell but not directly in INs. An alternative, more general approach would be to write programs using a generic ListCase agent of arity 5, whose behaviour is defined by the two following rules.
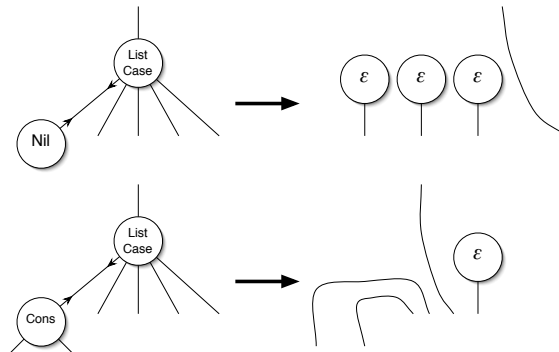
The idea is that two different nets are connected to the ListCase agent. One is a net to be returned when the argument list is empty, and the other is a net with three ports, used to combine the head and tail of a non-empty list. Observe that one of these nets is not used in each rule, and must be erased with ε agents.

This approach can be followed for encoding higher-order functions in general. In fact, the ListCase agent corresponds to the following function definition:

```
listcase :: [a] -> b -> (a -> [a] -> b) -> b
listcase    []   x _ = x
listcase (x:xs) _ f = f x xs
```

In this example, the two nets connected to ListCase will play the role of f and x in the above Haskell version. In general, an argument of type $T_n \to \ldots \to T_1 \to T_0$ will give rise to n ports in the agent, to which an appropriate net is connected when the function is applied.

If we now want to encode app using the ListCase agent, we also need a way of encoding recursion. This is a classic problem in graph-rewriting implementations, and also in the field of term-graph rewriting. We use a simple solution here, considering that interaction rules may introduce new active pairs: recursive definitions are coded by adding explicit rules for a fixpoint agent called rec. In the present case this yields the following net, where a δ agent is used to *duplicate* a net:
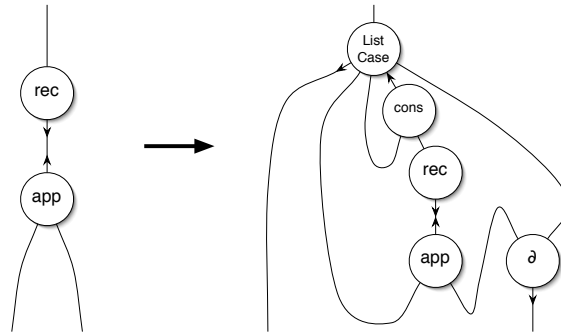
Now our program can be encoded as the following net, to be reduced in the context of the system containing the rules for the agent ListCase and the fixpoint rule for app.
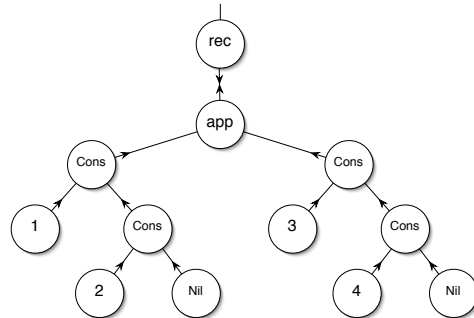
The following table summarizes our approach to Visual Programming:

Functional Programming	*Visual Functional Programming*
Function	Agent
Function Definition	Set of Interaction Rules of an Agent
Expression	Interaction Net

4 Agent Archetypes

Although no standard programming language exists for 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.
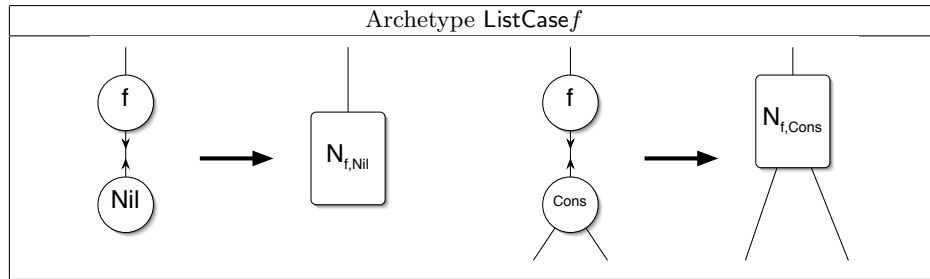
Fig. 2.

To illustrate what we mean, 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 by *archetype*. The ListCase archetype is defined in figure 2 and 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 any nets, called respectively $N_{f,\text{Nil}}$ and $N_{f,\text{Cons}}$.

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 expected interaction rules for the new agent in the interaction system being defined.

An example is given in figure 3, where the isZero agent is defined as an instance of the ListCase archetype. In this example, ε agents are used to *erase* the head and tail of the list, which are not used in the result.
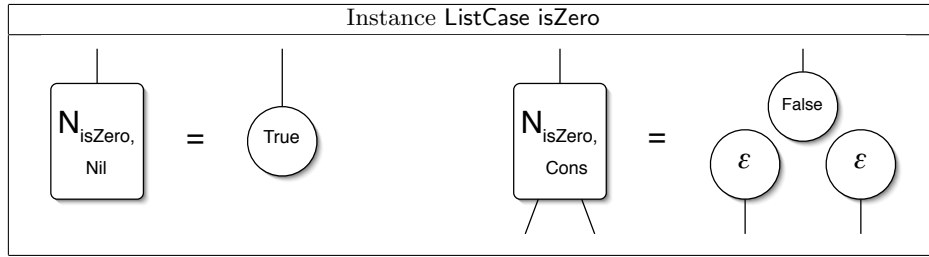
Fig. 3.

5 Programming with Folds

A fundamental aspect of Functional Programming 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 corresponding to *iteration* over the elements of the list, encoded by the following higher-order function:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

A function like `sum` is often called a *fold*, in the sense that its definition is such that it can be written using `foldr`. The use of recursion patterns has the advantage of being appropriate for program transformation and reasoning using the so-called *calculation-based* style. A classic example is the following *fusion* or *promotion* law [1], which states how the composition of a function with a fold over lists can be transformed into a single fold.

$$f \ (\text{foldr } g \ e \ l) = \text{foldr } h \ c \ l$$
if f is a strict function, $f \ e = c$, and $f \ (g \ x \ r) = h \ x \ (f \ r)$ for all x, r

One of the goals of this paper is to derive a visual version of this law and to give examples of its use in program transformation.

Interaction Net Programming with Folds. It is straightforward to identify the rules that characterize an agent corresponding to a fold. If we take the case of lists, interaction rules must be defined for f to interact with both Nil and Cons:

- interaction with Nil results in an arbitrary net;
- interaction with Cons sends an f agent along the tail of the argument list, and a net $N_{f,\text{Cons}}$ then combines the head of the list with the recursive result.

Following our discussion of programming with higher-order agents in section 3, we could now either define a single foldr agent with enough ports to connect nets corresponding to its arguments, or else define an appropriate archetype. We follow the latter approach, which leads to visually much simpler and more concise definitions.
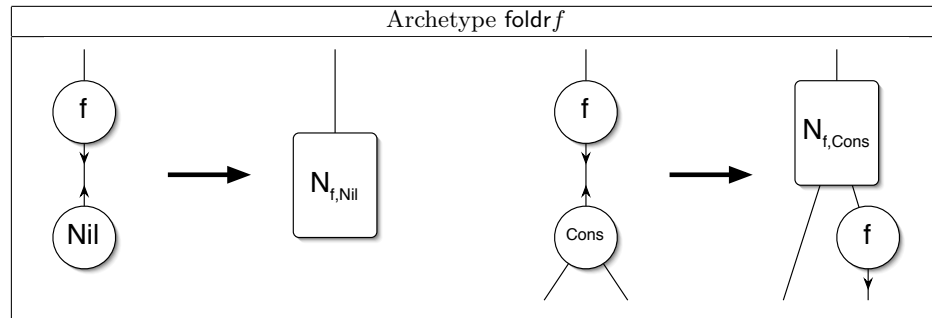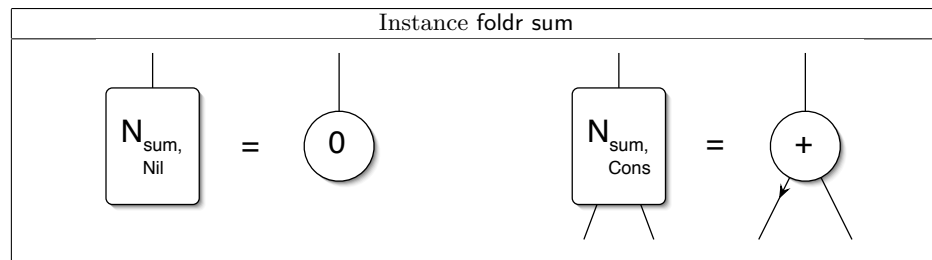

Fig. 4.

Fig. 5.

The foldr archetype is defined in figure 4. The archetype is recursive in the sense that the parameterized agent occurs in the right-hand side of one of the rules. As an example, the definition sum = foldr (+) 0 becomes the instance given in figure 5.

As a second example of a fold archetype, appendix A contains the definition of an archetype for folds over *Leaf-labelled binary trees*.

6   Higher-order Folds.

Our current definition of a fold agent is still not satisfactory, and will now be generalized. 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, since it iterates over its first argument to produce a
function that takes the second argument:

```
app = foldr (\x r l -> x:(r l)) id
```
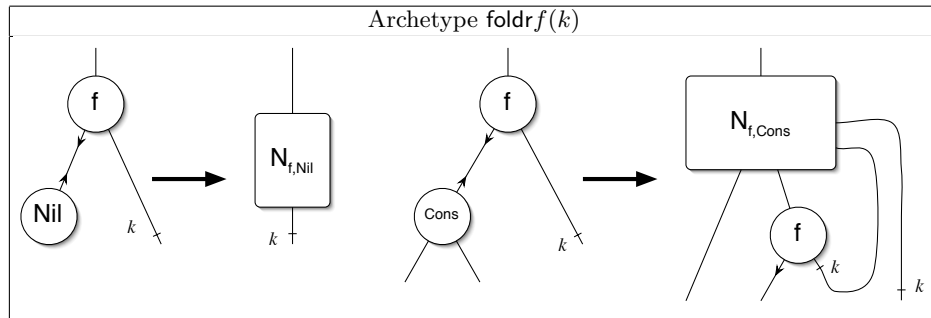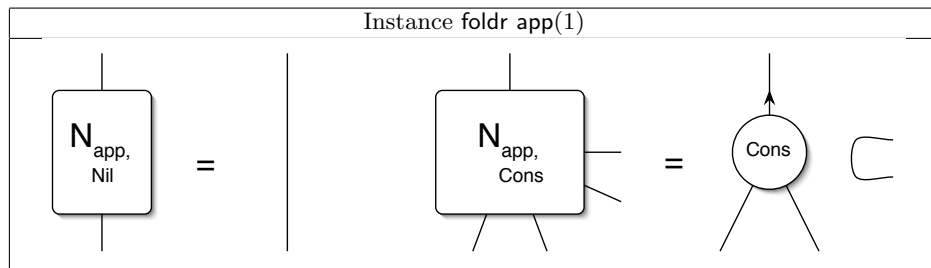

Fig. 6.

Fig. 7.

This fold can be defined with interaction nets as in figure 1, which clearly does
not match our current definition of the fold archetype. Functions of more than
one argument defined as folds over the first argument lead us to the generalization
of the foldr archetype shown in figure 6. 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 in figure 7. 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.

As a slightly more complicated example, take any tail-recursive function that uses an accumulator argument. For instance the following is an alternative way of calculating the sum of the elements in a list (invoked with an initial value of 0 for the accumulator):

```
sum' :: [Int] -> Int -> Int
sum'     [] y = y
sum' (x:xs) y = sum' xs (x+y)
```

and can also be written as `sum' = foldr (\x r y -> r (x+y)) id`.
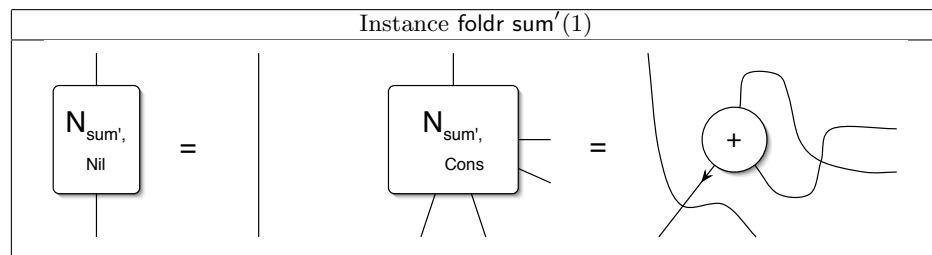
Fig. 8.

In interaction nets we have the instance of foldr shown in figure 8. There is again an open wire in the net $N_{\mathsf{prod'},\mathsf{Cons}}$, connecting the result of the fold to the result of the recursive call, which matches exactly the definition of tail-recursion.

7 Fusion

The goal of this section is to present a program transformation principle for visual functional programs, written as a *fusion law* for fold agents. This is the visual equivalent of the law mentioned in section 5. Although the law presented here applies to folds over lists, a similar appropriate law can be written for any regular datatype.

We remark that this transformation principle is stated in the Interaction Net framework (as an equivalence of nets) and may be proved using exclusively IN techniques. The law is closely related to the fusion law for functional programs, however no formal statement on this fact is made, since our encoding of functional programs is informal.

Proposition 1. *Let g, f, f' be agents of arity $k+1$, $j+1$, $j+k+1$ respectively, with f, f' defined as list fold agents:* Instance foldr $f(j)$, Instance foldr $f'(j+k)$. *Then we have*

$$g \circ f \quad \equiv \quad f'$$

if
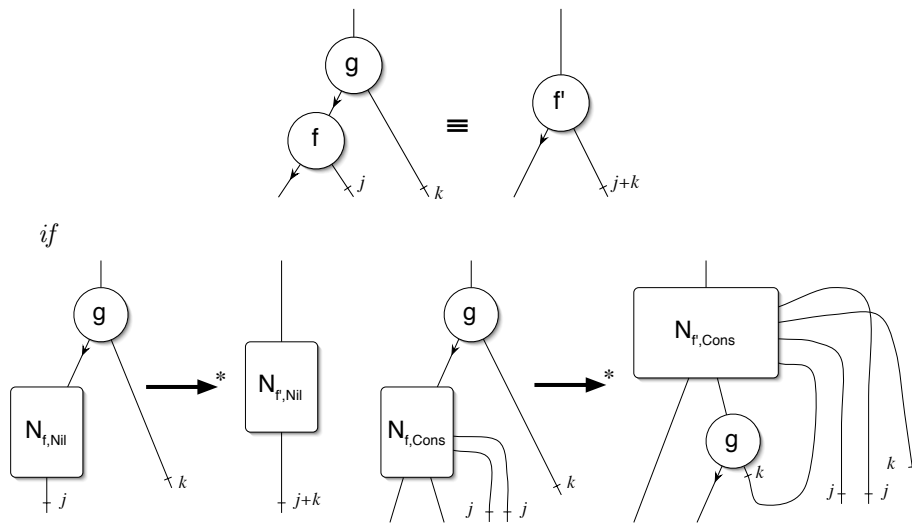
where $\longrightarrow^*$ stands for the transitive closure of reduction up to contextual equivalence.

Proof (sketch). Two proofs can be given, using different sets of assumptions. If one takes into account that the agents Nil and Cons are being used to implement a datatype defined as a least fixpoint (which is the case for finite lists), then a simple inductive proof can be given, on the structure of the argument list. This is the list connected to the leftmost port at the bottom of both nets in the equivalence. The base case is obtained by connecting a Nil agent; the inductive case is obtained by connecting the principal port of a Cons agent.

A more general proof uses a coinductive argument, following the principles mentioned in section 2. This proof works on the interaction system, disregarding the functional aspect and the structure of the datatype.

This proposition acts as a formal basis for the transformation of visual programs over lists, and can be used effectively as a transformation scheme. Its usefulness will now be illustrated by a number of examples.

Example 1. Our first example transforms a two-pass function into a single pass one. Consider adding the squares of the elements in a list with the composition of functions `sum . sqrs`, where `sqrs` maps the square function on a list. Then fusion can be used to transform this into a single fold. Visually we aim at the following transformation:
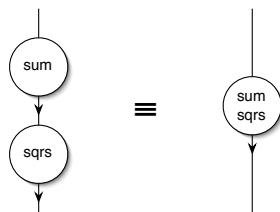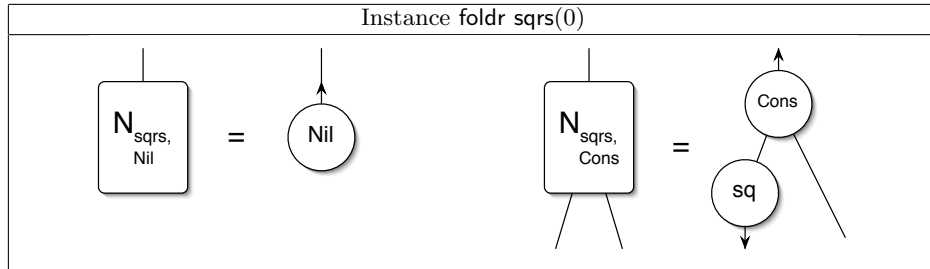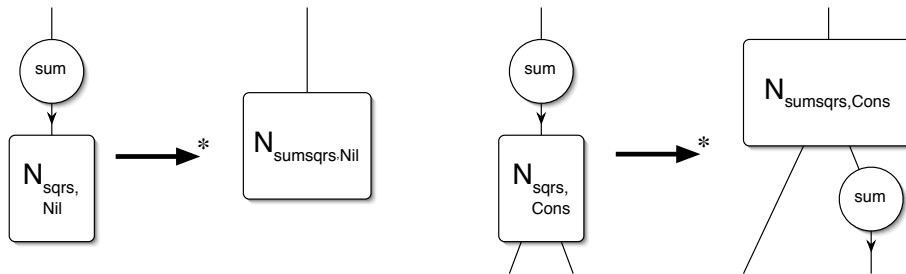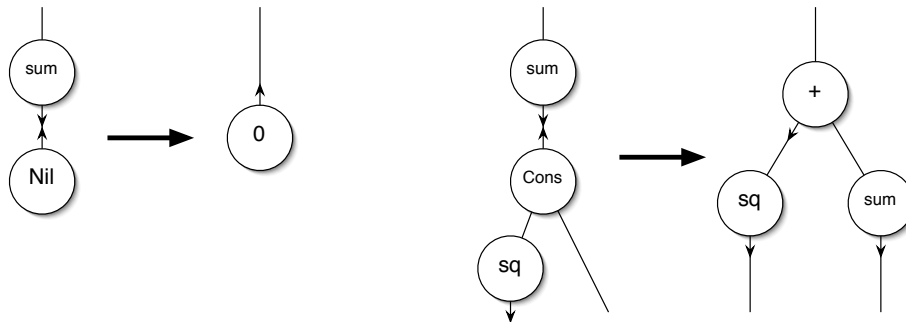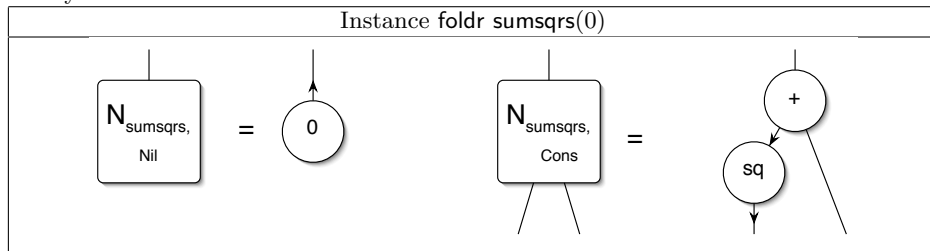
Fig. 9.

Figure 9 shows the definition of sqrs (where sq calculates the square of a number); the conditions for application of the fusion law (proposition 1) with $j = k = 0$; and the final definition obtained for sumsqrs. The transformation is straightforward and yields as result the expected agent definition.

Example 2. Our second example consists of deriving an accumulator-based optimization of the list-reversal function. To do this we depart from a *specification*. Let us call the single-argument reverse function rev, and the two-argument optimized version rev′. Their types are

```
rev  :: [a] -> [a]
rev' :: [a] -> [a] -> [a]
```

where the second argument of rev′ is an accumulator. Our specification is written as app (rev l) y = rev′ l y. Visually:
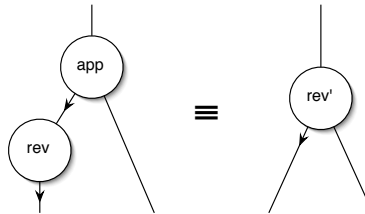
which states how the two versions are related.

Figure 10 shows the initial quadratic-time definition of rev; the conditions for application of fusion (proposition 1) with $j = 0$ and $k = 1$; and the final linear-time definition obtained for rev′, which redrawn here in a clearer way:
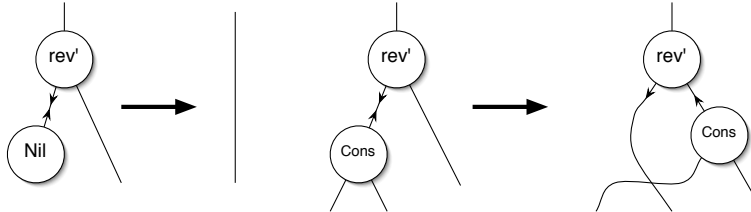

This corresponds to the Haskell definition

```
rev' [] y = y
rev' (x:xs) y = rev' xs (x:y)
```

We thus obtained a higher-order fold resulting from the fusion of a higher-order function with a first-order fold. Our next example produces a higher-order fold from the fusion of a first-order function with a higher-order fold.
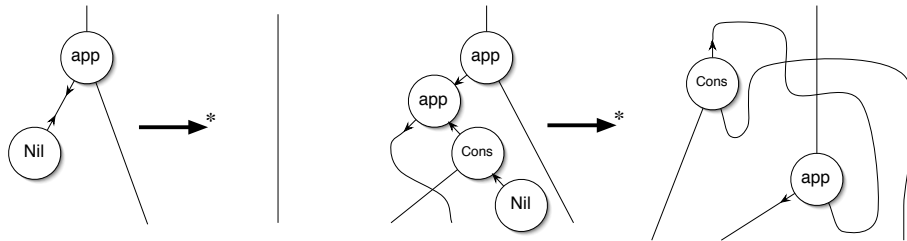
Example 3. Consider the ordered insertion of an element in a sorted list. This is defined as a fold in figure 12, top, with the behaviour of sort2 agent that sorts two elements specified in figure 11. Note that this specification does not commit us to any specific implementation of the elements being compared or of comparison
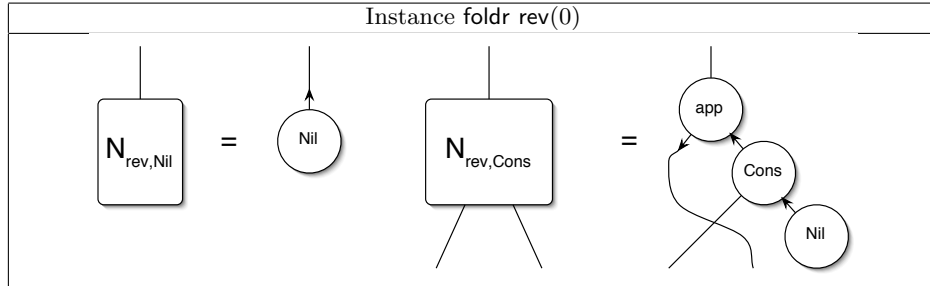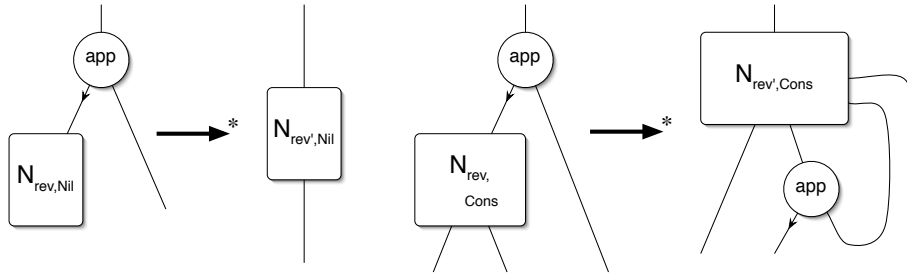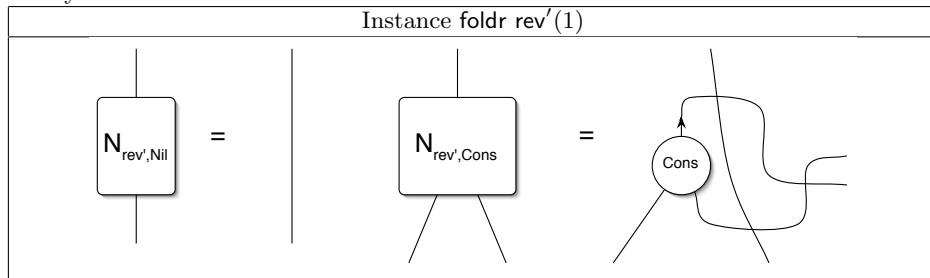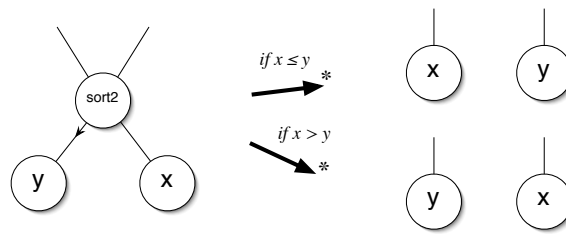
Fig. 10.

Fig. 11.

itself. One can imagine that elements belong to some basic type built into the system, and comparison is performed by a low-level call.

The definition of insert compares the head of the list with the element to be inserted. The smallest of these elements will be the head of the returned list. The tail is obtained by inserting the other element in the tail of the original list, which corresponds to a fold definition. Note that other definitions are possible; this example illustrates that some agents that would not immediately be defined as folds *can* be so defined, making possible the application of fusion.

Now consider taking the head of the list resulting from such an insertion. This will give as result the minimum element among those in the list and the element inserted. This can be transformed into a single fold agent:
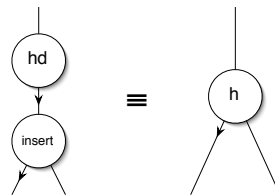

Figure 12 also shows the conditions for application of the fusion law (proposition 1) with $j = 1$ and $k = 0$, and the final definition obtained for h. We remark that the definition of hd on an empty net may be treated by introducing a special agent $\perp$, which gets erased by ε as any other agent (this applies to any partial function). This allows us to prove that a net consisting simply of an ε is contextually equivalent to an ε connected to a hd agent, which in turn can be used in order to identify an occurrence of hd connected to $N_{h,Cons}$.

8 Future Work

We are currently working on the implementation of a tool for visual functional programming based on these ideas. The tool will incorporate an interaction engine and an archetype definition and instantiation mechanism, and will allow the user to (visually) define functional programs (including new inductive datatypes); animate the execution of visual programs; and perform program transformations by fusion.

Instance foldr insert(1)
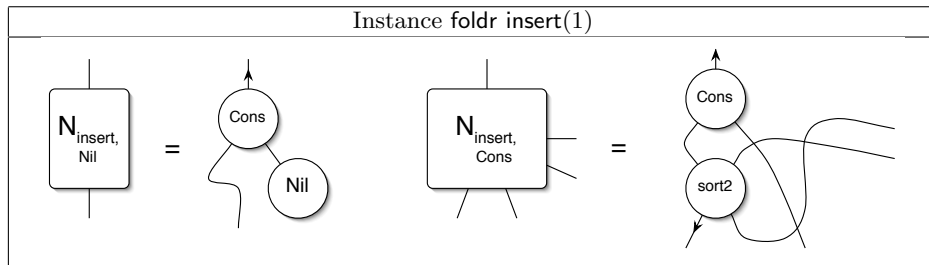
Conditions for fusion:
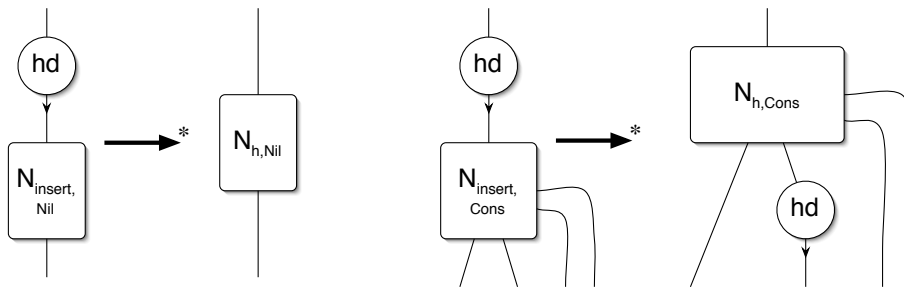
Substituting and reducing using definition of hd (returns head of list and sends an ε agent along the tail) and contextual equivalence:

This yields:
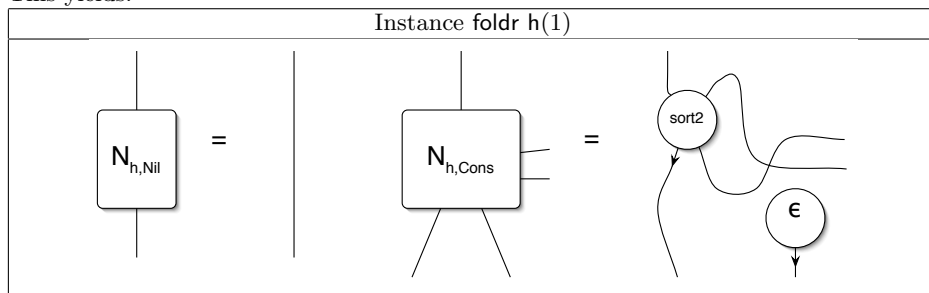

Instance foldr h(1)
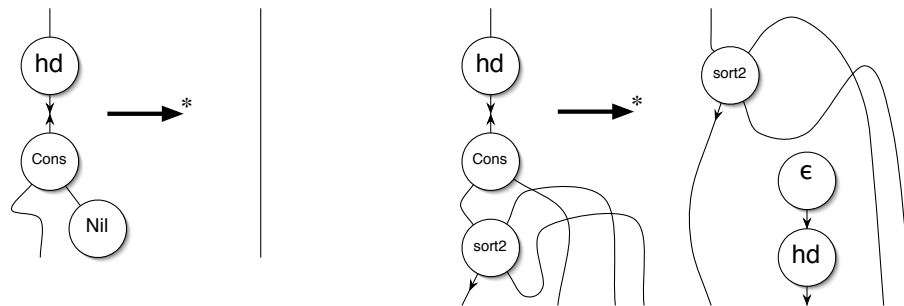
Fig. 12.

On the theoretical side, it is straightforward to derive visual fusion laws for folds over any regular type; it remains to study in this context the dual notion of an *unfold* archetype, together with the corresponding fusion law. Unfolds produce values of datatypes defined as *greatest fixpoints*; fusion is here not simply a systematization of structural induction, but an alternative proof-method to the more sophisticated fixpoint induction technique. In the IN framework, coinduction will certainly have to be used.

It should be mentioned that the use of fusion laws is largely used in the field of Datatype-generic Programming (see [2] for an introduction). The use of algebraic machinery allows for a unique definition of a fold (or unfold), with the datatype as parameter. Fusion laws are also generic in this sense, and can be instantiated for particular data-types. It remains to see how this generic aspect can be brought to the present visual framework. It would be useful to be able to derive fusion laws automatically for each new datatype from a generic scheme, and to incorporate this ability in the visual programming tool.

The notion of *hylomorphism* (the composition of a fold with an unfold in a language where least and greatest fixpoints coincide) clearly deserves to be studied in this framework, since a number of specific program transformations have been proposed for these functions.

References

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. J. Gibbons. Calculating Functional Programs. In *Proceedings of ISRG/SERG Research Colloquium*. School of Computing and Mathematical Sciences, Oxford Brookes University, 1997.
3. K. Hanna. Interactive Visual Functional Programming. In S. P. Jones, editor, *Proc. Intnl Conf. on Functional Programming*, pages 100–112. ACM, October 2002.
4. J. Kelso. *A Visual Programming Environment for Functional Languages.* PhD thesis, Murdoch University, 2002.
5. 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.
6. 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.
7. H. J. Reekie. Visual Haskell: A first attempt. Research Report 94.5, Key Centre for Advanced Computing Sciences, University of Technology, Sidney, Aug. 1994.

A Folds over Leaf-labeled Trees

Figure 13 contains the definition of an archetype for folds over binary trees with labeled leaves. This would be defined in Haskell by

```
data LTree a = Lf a | Nd (Ltree a) (LTree a)
```

The figure also shows a simple example of such a fold: the agent that traverses a tree from left to right, producing a list.
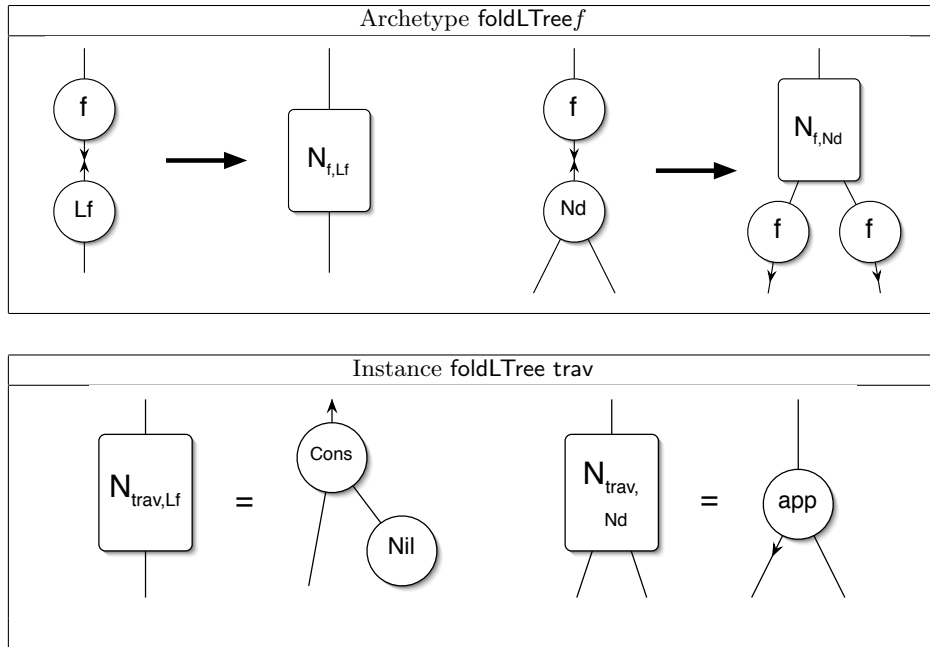

Fig. 13.