Electronic Communications of the EASST
Volume 38 (2010)

Proceedings of the Fifth International Conference on Graph Transformation - Doctoral Symposium (ICGT-DS 2010)

Realizing Impure Functions in Interaction Nets

Eugen Jiresch

17 pages

Guest Editor: Andrea Corradini
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Realizing Impure Functions in Interaction Nets

Eugen Jiresch*

Institute for Computer Languages
Vienna University of Technology

Abstract: We propose and illustrate first steps towards an extension of interaction nets based on monads to handle functions with side effects (e.g., I/O, exceptions). We define three monads for common types of side effects and show their correctness by proving the monad laws.

Keywords: interaction nets, side effects, monads

1 Introduction and Overview

Programming languages are the key to use computational resources efficiently. To ensure correctness and productivity of programs, formal verification of software has become increasingly important in the last years. Hence, programming languages based on rigorous formal models are indispensable. A prominent example is Haskell, a functional programming language based on the λ-calculus.

Interaction nets (INs for short) are a programming paradigm based on graph rewriting. One main idea of IN programs is to represent programs as graphs (nets). Their execution is modeled by rewriting the graph based on specific node (agent) replacement rules. This simple system is able to model both high- and low-level aspects of computation. The theory behind interaction nets is well developed. They enjoy several useful properties such as strong confluence and locality of reduction. These ensure that single computational steps in a net do not interfere with each other, and thus may be executed in parallel. Another important aspect is that interaction nets share computations: reducible expressions cannot be duplicated, which is beneficial for efficiency in computations.

While the properties above demonstrate great potential for interaction nets, the existing prototype languages based on interaction nets lack important features for practical use, such as input/output functionality, state manipulation or exception handling. Such features of interaction with the real world are often considered impure, as opposed to pure (mathematical) functions which do not incorporate side effects. Interaction net programs (or systems) can be viewed as a pure language: the reduction of a net is not influenced by anything but its initial state.

Nowadays, software frequently interacts with the real world. Impure functions are considered a vital part of programs, hence pure languages need to incorporate them. Thus, they require an appropriate interface to deal with impure effects.

To combine pure functions and the outside world in this fashion, we adapt a monad based model for interaction nets. Monads are a framework to structure computation that has been

* The author was supported by the Austrian Academy of Sciences (ÖAW) under grant no. 22932 and by the Vienna PhD School of Informatics.

used e.g. in Haskell with great success. However, there are two main obstacles when adapting monads to interaction nets: First, interaction nets need to support abstract datatypes in order to implement monads in a general way. Existing type systems ([2, 3]) for interaction nets are lacking this feature. Second, monadic functions have a distinct higher-order character. Even though interaction nets seem well-suited to incorporate higher-order functions (both data and computation are represented as nets and treated equally), as far as we know no model for higher-order interaction nets exists to date.

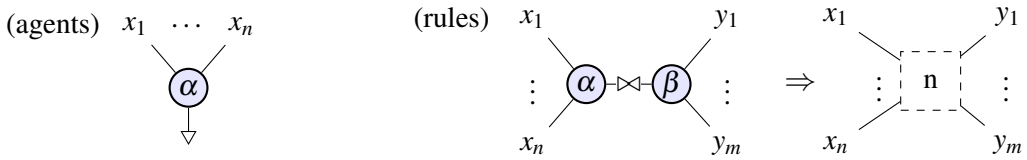
In this paper, we present three ad-hoc solutions to typical side effect computations in interaction nets based on monads. We show that these examples satisfy the monad laws. We then discuss the obstacles towards a more general side effect model based on the monad framework and present ideas on how to overcome them, which is currently work-in-progress. To summarize, the main contributions of the paper are:

- The definition of three monads, *Maybe*, *Writer* and *List* in interaction nets.
- A proof that the monad laws are satisfied by our implementation.
- An assessment of the remaining challenges towards a general solution.

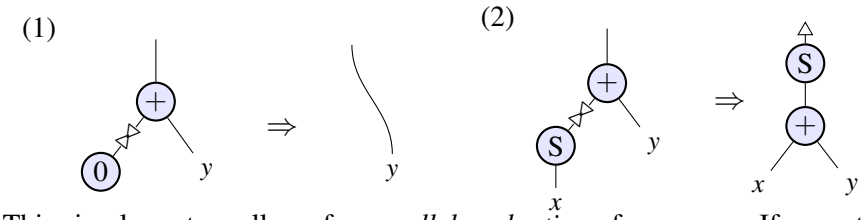
This paper is organised as follows: The remainder of this section gives a short introduction to interaction nets. In Section 2, we discuss computational side effects and the challenges they represent for pure languages. In Section 3, we discuss our approach to handle side effects in interaction nets. We develop three examples of monadic side effect handling and show their correctness. In Section 4, we discuss open problems - appropriately extending existing interaction nets type systems and handling rules with arbitrary/variable agents. Finally, we present a conclusion and give an outlook on further research.

1.1 Interaction nets

Interaction nets have been introduced in [9]. A *net* is a graph consisting of *agents* (nodes) and *ports* (edges). Computation is modeled by rewriting the graph, which is based on *interaction rules*. These rules apply to two nodes which are connected by their *principal ports* (denoted by the arrow). We refer to these nodes as *active pair* or *redex*. Interaction rules preserve the *interface* of the net: no auxiliary ports are added or removed.



For example, the following rules model the addition of natural numbers (encoded by 0 and a successor function S):



This simple system allows for *parallel evaluation* of programs: If more than one interaction rule is applicable at the same time, they can be applied in parallel without interfering with each other. In addition, active pairs in a net cannot be duplicated: They are evaluated only once, which allows for *sharing of computation*. Only parts of a net without active pairs (i.e., in normal form) can be duplicated.

However, functions that incorporate side effects such as I/O or exception handling generally destroy the above properties. Yet, these impure functions are a vital part of any programming language. Our goal is to extend interaction nets in order to support computations with side effects.

2 Computational Side Effects and Impure Functions

A *pure* (mathematical) function's result is determined only by its input parameters. While this definition may seem trivial at first glance, many functions and procedures do not qualify as pure functions. The state of the machine executing the program and interaction with the user or other devices can have considerable impact on a function's result. *Impure* functions can be influenced by computational side effects (such as a change in machine state) or cause side effects themselves in addition to their return value.

To elaborate on this difference, consider for example the following function:

```
def square(x):
    global y
    y = y+1
    return x*x
```

The function *square* computes the square of its input parameter and returns it as a result. In addition to this, it accesses a global variable y (being a part of the state of the program) and increments it. This is a typical example of a computational side effect that occurs next to the "actual" result of the function.

However, we can define a pure function that has almost the same behaviour as *square* by adding additional parameters:

```
def square_inc(x,y):
    return (x',y') where
        x' = x*x
        y' = y+1
```

The second argument of *square_inc* replaces the global variable y. Furthermore, the function returns a pair consisting of the square of x and the incremented y. This way, its behaviour (i.e., its result) is determined only by its input parameters.

Programs that consist only of pure functions have a clear advantage: They allow for *equational reasoning* - also referred to as "substituting equals for equals for equals" - which is a fundamental principle of mathematics. It allows one to state properties of programs such as their correctness more easily, which is much more difficult in the presence of side effects. Therefore, it is advantageous to encode side effects with additional parameters. However, doing so essentially transfers the problem to the programmer: handling additional arguments and "plumbing" them through the flow of the program is tedious and error-prone. A general and abstract solution to handle computational side effects in a pure environment is therefore needed.

3 A Monad Approach to Side Effects

We base our solution to impure functions in interaction nets on *monads* [11]. Monads are a model to structure computation. They have been used with great success in functional programming: monadic functions can encode side effects in a pure language and determine the order in which they occur [13].

Formally, a monad consists of an abstract datatype *M a* and two (higher-order) functions operating on this type: [1]

```
data M a
return       :: a -> M a
>>= (bind)  :: M a -> (a -> M b) -> M b
```

Monads are used to augment values of some type with computations that contain potential side effects. Intuitively, a monad does the following:

- *M* adds a sort of wrapper to some value *x* of type *a*, potentially containing additional data or functions.
- *return x* wraps a value *x* without any additional computations.
- *bind* handles sequentialization or ordering of function applications and their potential side effects.

A monad needs to satisfy the following laws:

```
(1)    return a >>= f       = f a
(2)    m >>= return         = m
(3)    (m >>= f) >>= g      = m >>= (\x -> (f x >>= g))
```

Intuitively, *return* performs no side effects. Law (1) states that if *return a* is the first argument of *bind*, its result should be equal to the application of its second argument to *a* (without any side effects). According to law (2), *return* also acts as a right neutral element for *bind*. In turn, *bind* has a property that is similar to associativity. This is expressed by law (3).

Interaction nets as such do not support abstract datatypes or higher-order functions, which are essential ingredients of monads. This, together with the restricted shape of rules, makes an

[1] The monad functions and laws are expressed in a syntax similar to Haskell.

adaptation non-trivial. However, we can define a monad just by the basic features of interaction nets, agents and rules. We will use the following agents to model *return* and *bind*:

The argument of *return* (a base value) is connected to *ret*'s principal port. The principal port of $>>=$ is connected to a wrapped value. Its auxiliary port is connected to an agent representing *bind*'s second argument, a function.

While *bind* may be modeled differently (e.g., more auxiliary ports), this approach captures the essence of monadic side effect handling in a natural and intuitive way. Moreover, it allows us to conveniently show one of the monad laws in the next subsection.

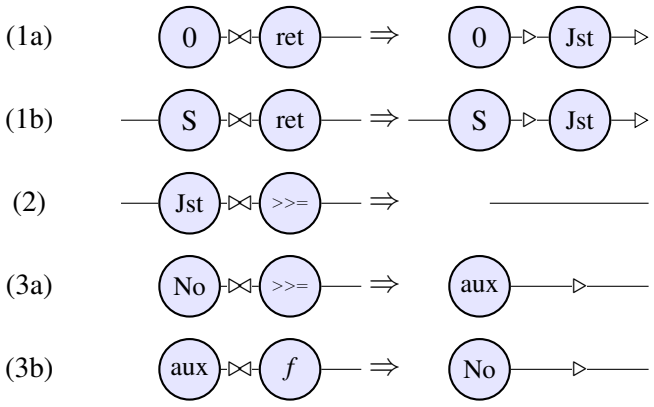
3.1 The Maybe Monad

We illustrate the basic idea and approach by a typical example, namely the monad *Maybe* which is used for exception handling:

```
        data Maybe a    = Just a | Nothing
(1)     return x        = Just x
(2)     (Just x) >>= f  = f x
(3)     Nothing  >>= f  = Nothing
```

The idea behind the maybe monad is simple: Any function capable of exception handling either returns a regular value (Just a) or a special value denoting an exception (Nothing). *bind* is used to concatenate two functions of this form: If the result of the first function is a regular value, it is simply passed to the second function. If the result of the first function is Nothing, *bind* also returns Nothing, while effectively discarding its second argument. The *Maybe* monad satisfies the monad laws. This can easily be verified by substituting the variables of the equations with *Nothing* or *Just* values.

The following rules model the Maybe monad for functions f on natural numbers:

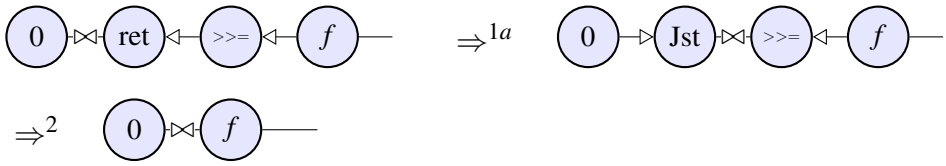


The correspondence of interaction rules and original definition of the Maybe monad is indicated by the rule labels. Note that rule (3) of the Maybe monad is split in two interaction rules, introducing an auxiliary agent. The reason for this is that the left-hand side (LHS) of an interaction rule may only consist of two agents. However, the LHS of rule (3) consists of three

non-variable symbols (Nothing, f, >>=). This requires the rule to be split in two. The restriction of two agents per rule LHS can be relaxed, as we will see in the writer or list monad examples. However, for the sake of clarity, we define the maybe monad with ordinary interaction rules only (i.e., two agents per LHS).

3.1.1 Correctness of the Maybe Monad

The interaction rules above satisfy the monad laws, which can be shown by reduction sequences of nets. We write $\Rightarrow^r$ for a reduction step by applying rule r. The following sequence proves law (1) (return a >>= f) = f a:



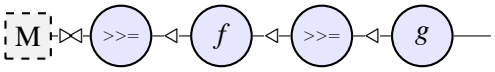
We only show the case $a = 0$. For $a > 0$, law (1) can be shown in the same way.
Law (2) is proved by the following sequence: (for m = Just 0)

Again, the case m = Just (S x) is proved similarly. Note that the case m = Nothing follows trivially from the definition of rules (3a) and (3b).

We can show the monad law (3) in a more general way. In fact, the following proof will hold for all three monads defined in this paper. Our main argument is that if an agent with one auxiliary port is used to model *bind*, law (3) is always true.

Proposition 3.1.2 *If an agent with only one auxiliary port is used to encode >>= in interaction nets, then the monad law (3) (m >>= f) >>= g = m >>= (\x -> (f x >>= g)) holds.*

Proof. We use an agent with one auxiliary port to model >>=. This way, the interaction net representation of each side of the above equation is the same, namely (the arbitrary net M is denoted by a dashed square):



Therefore, it is trivially true. □

So why do the interaction rules above constitute a monad? Two reasons can be given: First, we have defined agents and rules that have the same functionality as the *Maybe* monad: The agents *Jst* and *No* model the abstract datatype Maybe a. The agents *ret* and >>= have a behaviour equivalent to the original monadic operators. Therefore, these rules can effectively be used to model exception handling in interaction nets. Second, as we have shown above, the monad laws hold.

3.2 The Writer Monad

The Writer monad adds the possibility for functions to have a secondary, optional output. This output is accumulated through the evaluation of the program, combining the result of individual functions. As the name suggests, the Writer monad generalizes logging or debugging output. In addition to a programs result or actions, it provides information on how the result was achieved, whether any errors occurred or how much time the computation took.

The definition of the Writer monad builds on a type S used for the secondary output and a function combining values of this type, which we denote by *. In order to satisfy the monad laws, it is required that S and * form a *monoid*. Most commonly, the type *string* (with the empty string as the identity element) and a function for string concatenation is used, which clearly satisfies the monoid laws.

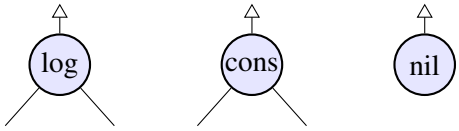
Formally, we define the Writer monad as follows, where e denotes the identity element of S:

```
        data Log a   = (a, S)
(1)     return x     = (x,e)
(2)     (x,s) >>= f  = (y,s*s') where (y,s') = f x
```

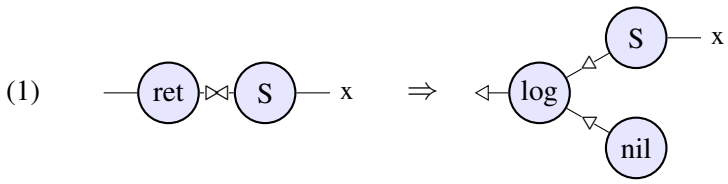
The monadic type *Log* is simply a pair of the base type a and the output type S. Here, *return* simply yields a pair of its argument and the identity element of S, *bind* applies f to x and returns a pair of the primary result of f and a combination of f's secondary and previous outputs.

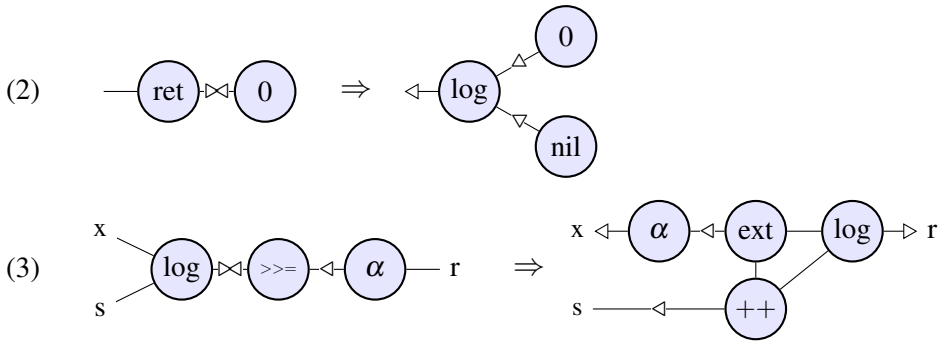
3.2.1 Adaptation to Interaction Nets using Nested Patterns

As with the Maybe monad, we restrict ourselves to the definition of monadic rules for concrete types. As primary return type a, we again use symbolic natural numbers. For the sake of simplicity, we use a list of booleans as secondary output type and regular list concatenation as combining function (denoted by ++). The agent *log* models the pair of return type (connected to the left auxiliary port) and secondary output (connected to the right auxiliary port). The agents *Cons* and *Nil* are used to express lists.



The interaction rules for the Writer monad are similar to the definition above. In the rule for *bind*, we use the agent *ext* to "extract" both values of the *log* pair that is returned by the generic agent α.





Note that rule (3) (modelling bind) has three agents in its LHS. This clearly violates the restriction of two agents per LHS. However, this restriction can be overcome in the form of *nested patterns*. Nested patterns first appeared in [5]. We now recall the main definition of nested patterns and the conditions for preserving strong confluence.

Definition 3.2.1 (Nested active pair [5]) *A nested active pair is defined as follows:*

- *a regular active pair is a nested active pair, represented textually as* $\langle \alpha(x_1,\ldots,x_n) \bowtie \beta(y_1,\ldots,y_m)\rangle$ *(*x_i *and* y_j *represent auxiliary ports).*
- *if P is a nested active pair, connecting the principal port of some other agent to some auxiliary port* y_j *of P also yields an active pair. We represent this textually as* $\langle P, y_j - \gamma(z_1,\ldots,z_n)\rangle$.

The framework of interaction rules with nested active pairs is called INP (**I**nteraction nets with **N**ested **P**atterns). Since rules in INP may overlap due to the more complex LHS patterns, a condition to preserve strong confluence is introduced.

Definition 3.2.2 (Sequential set property [5]) *A set of nested active pairs N is* sequential *if and only if, when* $\langle P, y_j - \gamma(z_1,\ldots,z_n)\rangle \in N$, *then*

- *for the nested pair P,* $P \in N$
- *for all free ports y in P except* y_j *and for all agents* α, $\langle P, y - \alpha(z_1,\ldots,z_n)\rangle \notin N$

Intuitively, nested pairs in a sequential set do not overlap unless one is a subnet of another.

Definition 3.2.3 ([5]) *A set of rules R is* well-formed *if and only if*

- *there is a sequential set which contains every nested active pair of all LHSs in R*
- *for every rule* $P \Rightarrow N \in R$, *there is no rule* $P' \Rightarrow N' \in R$ *such that* P' *is a subnet of P.*

Well-formedness of a set of rules is sufficient for strong confluence: [2]

[2] More precisely, by strong confluence we here mean the property: Whenever $M \Leftarrow N \Rightarrow P$ with $M \neq P$, then there exists a net Q s.t. $M \Rightarrow Q \Leftarrow P$

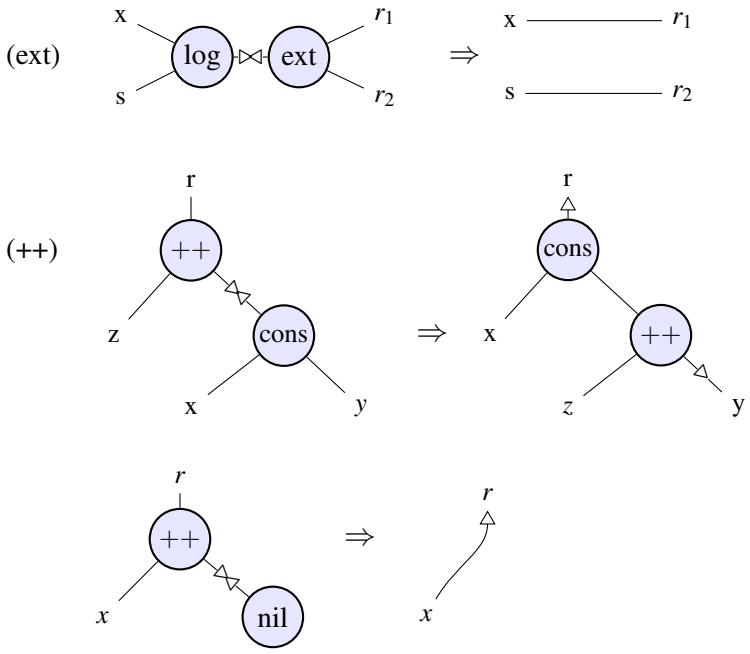
Proposition 3.2.4 ([5]) *If a set of rules R in INP is well-formed, then the reduction relation induced by R is strongly confluent.*

Using Proposition 3.2.4, we show that the rules for the Writer monad are strongly confluent.

Proposition 3.2.5 *The set of rules of the Writer monad is strongly confluent.*

Proof. The set of rules is well-formed: The active pairs of all LHSs are distinct. Therefore, no LHS is a subnet of another, and all LHSs can be added to a sequential set. Hence, by Proposition 3.2.4, the reduction using the rules of the Writer monad is strongly confluent. □

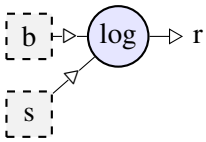
To handle the access to the elements of the *log* pair and concatenation of lists, we define the following auxiliary rules.



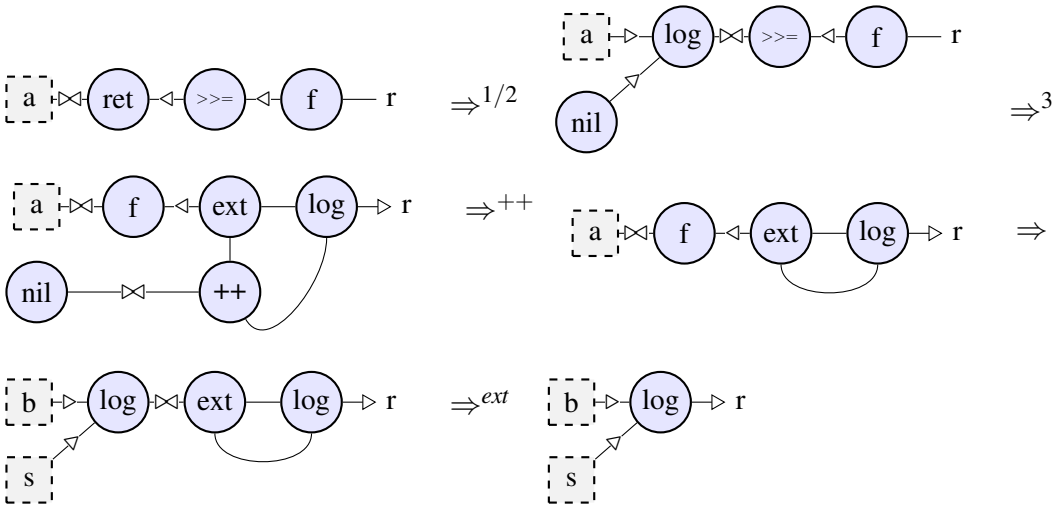
The *ext* agent extracts both elements of the log pair and returns it via its two ports r_1 and r_2. ++ performs the basic concatenation of two lists.

3.2.2 Correctness of the Writer Monad

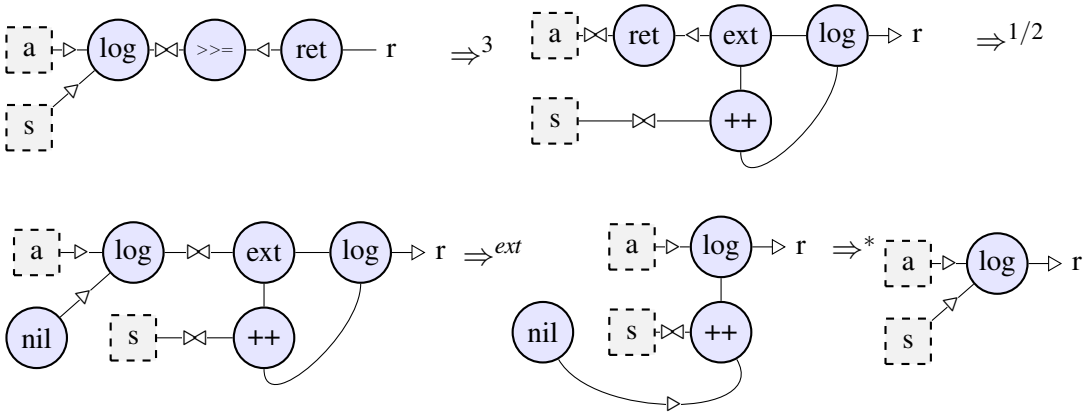
Similar to the maybe monad, we can show the correctness of the writer monad by reduction sequences of the terms on each side of the respective monad law. We begin with law (1), *return* $a >>= f = f\,a$. Let a, b be arbitrary but fixed natural numbers such that $f\,a$ = *(b,s)* (where s is the log information of f). We can then show that the interaction net encoding of *return* $a >>= f$ reduces to the one of *(b,s)*, namely



(the arbitrary nets a, b and s are represented by dashed squares):



The first step uses rule (1) or (2) depending on the concrete instance of a (0 or $S(x)$).
The following reduction sequence shows law (2), *(m >>= return) = m* (where *m = (a,s)*):



In the last step, the concatenation rule is applied multiple times (denoted by $\Rightarrow^*$) until a normal form is reached. It follows from the definition of the (++) rules that s ++ [ ] reduces to s.

As we use the same modeling approach as with the *Maybe* monad, the third monad law holds due to Proposition 3.1.2.

3.3 The List Monad

The List monad is used to sequentialize functions that take a single value as input and return an ordered sequence of values as output. For example, consider two functions f :: a -> [b]

and g :: b -> [c], where [t] denotes a list of type t. As each function takes just a single argument but returns a list, the operators of the list monad operators are used to concatenate them. In essence, the function g is simply applied to all results of f, combining all results of g's applications in a single list.

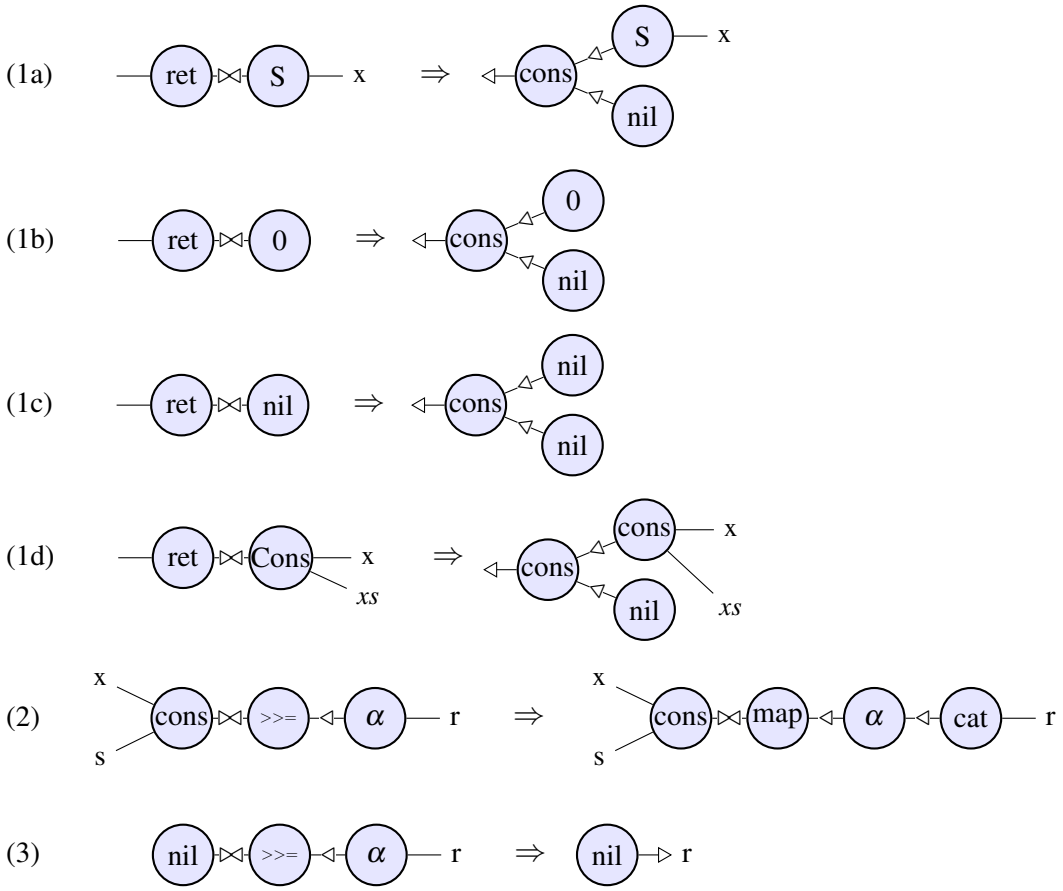
In Haskell, the List monad is defined as follows:

```
          data List a = [a]
(1)       return x = [x]
(2)       xs >>= f = concat (map f xs)
```

Here, *return* does nothing but wrapping a value in a singleton list. Furthermore, >>= applies the function f to every element of the input list and concatenates all resulting lists into a single list. *concat* and *map* are two well-known operators in functional programming: *concat* simply merges a list of lists into a single list; *map* has two arguments, a function and a list; *map* applies the function to every element of the list and returns the list of results.

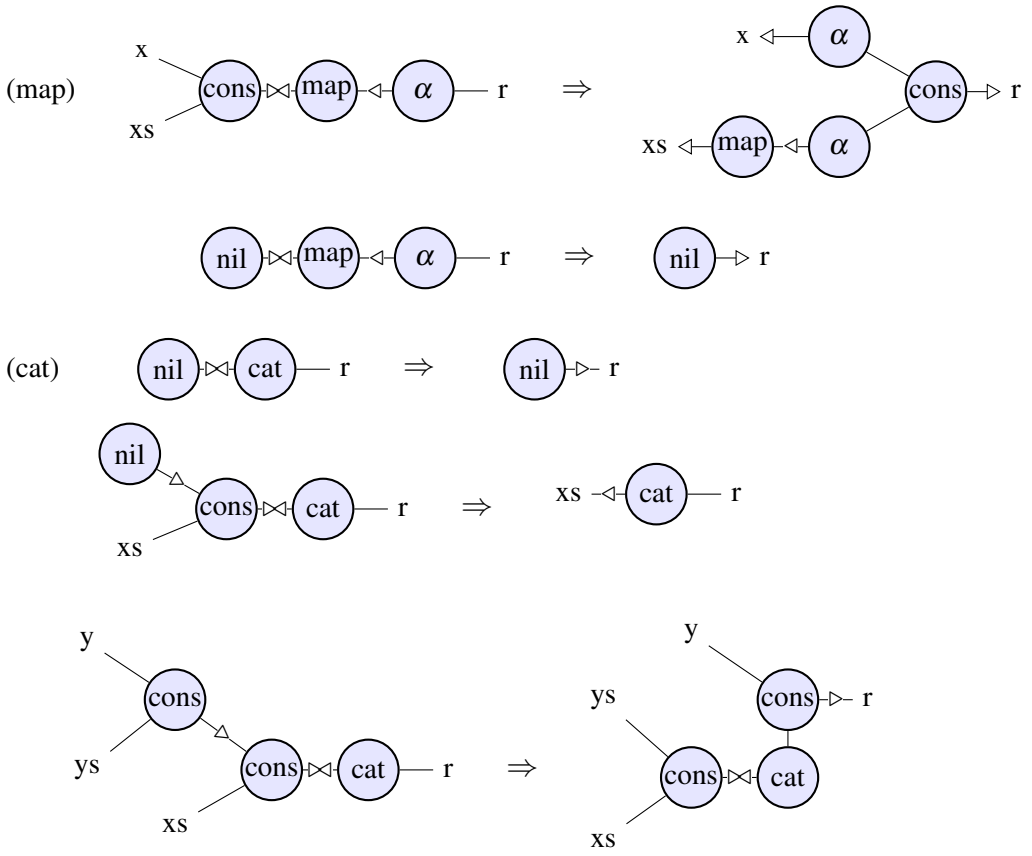
3.3.1 Adaptation to Interaction Nets using Nested Patterns

Instead of a generic type a, we allow only symbolic natural numbers and lists in the interaction net setting. We use the same agents to represent lists as we did for the Writer monad.



The rules for *return* are obviously very similar to the corresponding rules of the Writer monad. Instead of a pair of a value and an empty list, a singleton list is returned. The interaction rule for $>>=$ behaves analogously to the function definition in Haskell. Rule (3) models the behavior of *bind* when connected to an empty list. In this case, the empty list is simply propagated and the agent α is discarded.

In rule (2), the auxiliary agents *map* and *cat* are used. Their corresponding interaction rules are defined as follows:



The interaction rules for *map* have a very similar shape as the ones for *bind*. The second argument of *map* (an arbitrary function represented by the agent α) is connected to its auxiliary port. α is applied to the first element of the list and *map* is connected to the remainder of the list. The concatenate function, represented by *cat*, flattens a list of lists. The elements of each list are appended to a single list one by one.

Proposition 3.3.1 *Let R be the set of interaction rules consisting of the List monad, map and cat rules. The reduction relation induced by R is strongly confluent.*

Proof. R is well-formed: First, no LHS is a subnet of another. Second, all LHSs can be added to a sequential set. Note that only two rules share the same active pair (the second and third rule of (cat). However, these LHSs can be added to a sequential set as the respective nested agent (*nil* and *cons*) is connected to the same port. This satisfies the sequential set condition. Hence, by

Proposition 3.2.4, the reduction relation induced by R is strongly confluent. □

In addition, we show that all three monads can be added to a single ruleset without losing strong confluence. However, it is important to rename the monadic operators (and auxiliary functions) of each monad: without a suitable type system (see Section 4.2), we are unable to distinguish between the monadic operators of different monads.

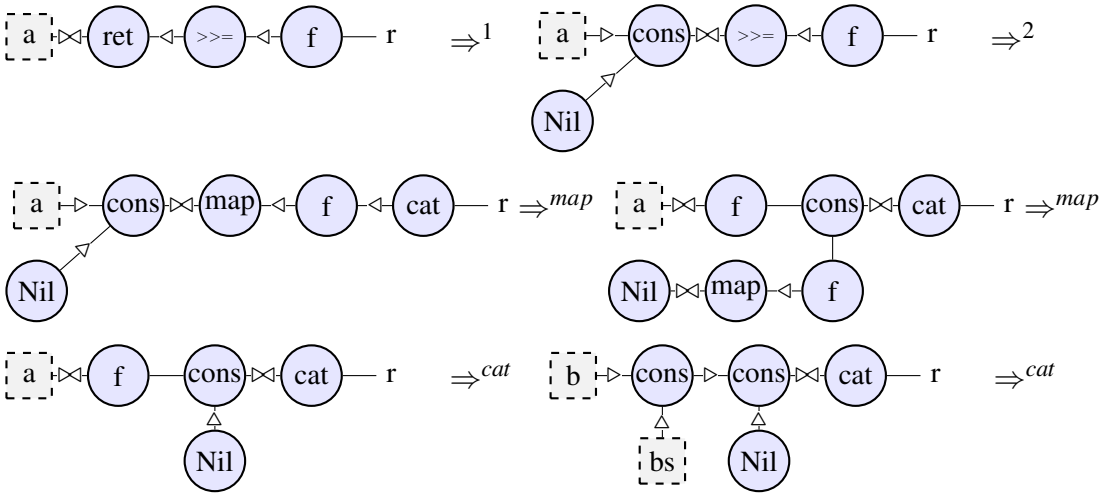
Proposition 3.3.2 *Let R be the set of interaction rules consisting of the Maybe, Writer and List monad including auxiliary agents, where the agents ret and >>= are renamed for each monad. The reduction relation induced by R is strongly confluent.*

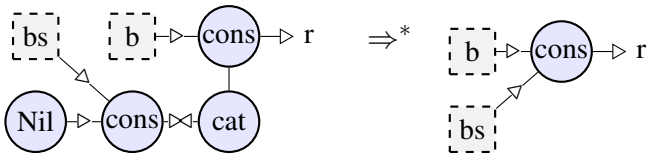
Proof. R is well-formed: No LHS is a subnet of another. In addition, all LHSs can be added to a sequential set: As shown before, the rules of the *Writer* and *List* monad can be added to a sequential set individually. The *Maybe* monad satisfies strong confluence as it consists of ordinary rules only. Furthermore, no two rules of different monads share the same active pair (as *ret* and >>= have been renamed). Therefore, R satisfies the sequential set condition. By Proposition 3.2.4, the reduction relation induced R is strongly confluent. □

Note that if the system consisting of these monads is added to other interaction rule sets, it is again required to verify well-formedness w.r.t. nested patterns for the resulting set of rules.

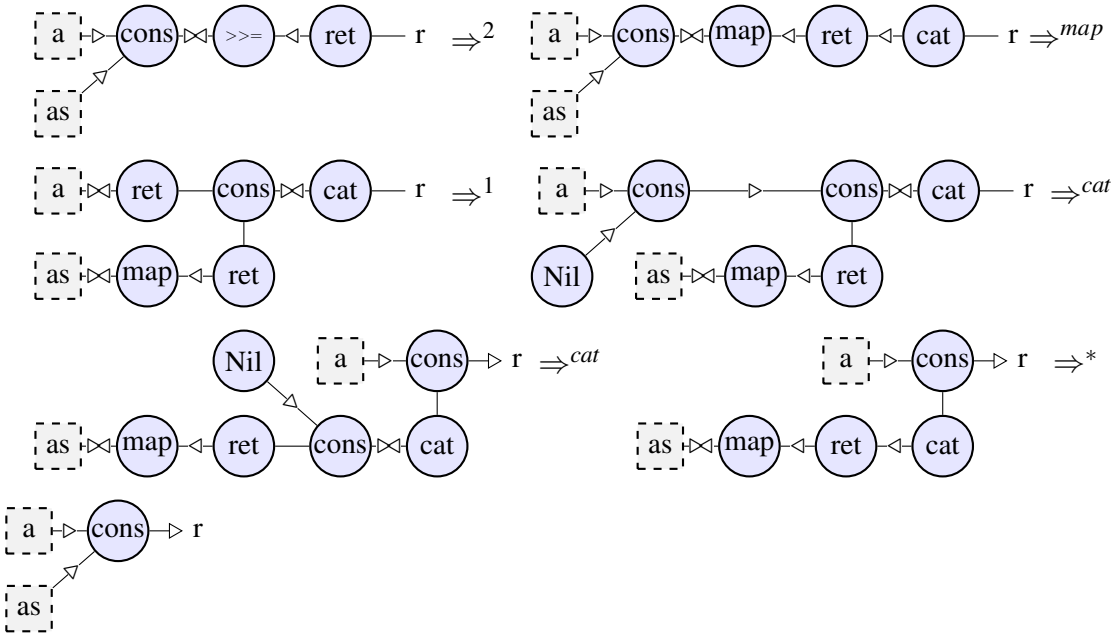
3.3.2 Correctness of the List Monad

We show the correctness of the *List* monad laws analogously to the previous examples. As for the Writer monad, we use arbitrary but fixed nets (denoted by dashed squares) in the reduction sequences showing the monad laws. Let a,b,bs be arbitrary nets such that $f\ a$ = *Cons* b bs. The following reduction shows law (1), *return* a >>= f = $f\ a$:





The last step is achieved by multiple applications of the (cat) rules (one for each element in *bs*). Law (2), *(Cons a as >>= return) = (Cons a as)* can be proved by the following reduction:



Again, the final step consists of several reductions. It follows from the definition of *map*, *cat* and *return* that *cat (map return as) = as*. In essence, every element in *as* is put in a singleton list. The resulting nested list is flattened again by *cat*.

As with the previous monads, law 3 holds due to Proposition 3.1.2.

4 Challenges

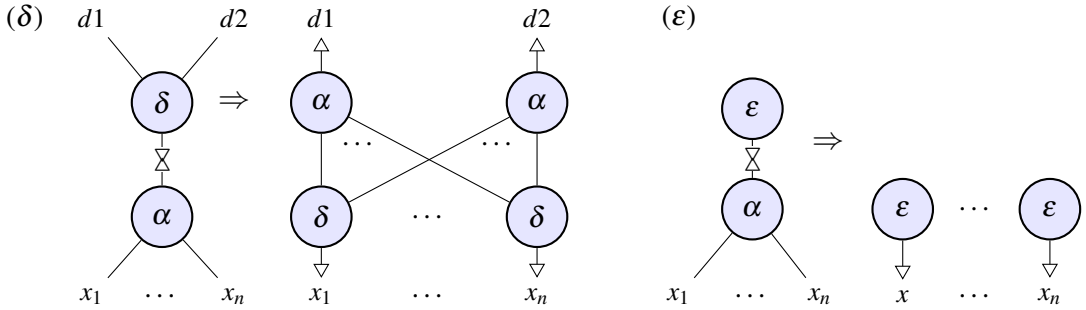
In this section, we discuss the main obstacles towards realizing a general monad framework for interaction nets.

4.1 Interaction Rules with Generic Agents

Monads have a distinct higher-order character. The second argument of the *bind* operator is a function itself. Furthermore, the monad laws contain variables that represent arbitrary functions. It is clear that interaction nets need to support these features.

In general, interaction nets seem well-suited for higher-order computations: data and functions are treated alike, as both are represented by agents. However, patterns of interaction rules consist only of specific agents. There are no variable or *generic* agents that would allow a specific agent

to interact with any agent in the same way. The exception to this rule are the agents ε and δ. These agents are frequently used in the literature (e.g., [10]) to model explicit duplication and deletion of agents. In fact, ε and δ interact in the same way with any agent, they delete (respectively, duplicate) the agent and propagate themselves to every auxiliary port:



In the same sense, we used α representing an arbitrary agent to specify some of the monadic rules in this paper. While ε and δ are frequently used, there is currently little theory on rules with variable or generic agents - these rules are usually just stated. Furthermore, no existing interaction nets implementation supports such rules in a general way.

We argue that this deficiency needs to be addressed in order to achieve a general and abstract framework of monads in interaction nets. For example, introducing generic rules yields several new possibilities of *rule overlaps*, i.e., more than one interaction rule matching a given active pair. Rule overlaps are not allowed in interaction nets by definition, and are very likely to destroy the confluence property. Therefore, a precise semantics of generic interaction rules needs to be defined, including constraints on their usage.

4.2 Type Systems

The second obstacle towards a general monad framework concerns the typing of interaction nets. Most of the monadic interaction rules in this paper are only defined for symbolic natural numbers, i.e., when interacting with the agents S or 0. Clearly, these rules should be defined for a range of types to represent a general solution. In particular, monads are usually defined for abstract data types with type variables, such as *Maybe a* or *List a*. Therefore, a suitable type system needs to support polymorphic types.

Several different type systems for interaction nets exist. For example, a basic system was defined by Lafont in the first paper on interaction nets [9]. The general idea behind it is to assign a type and a polarity (i.e., *input* or *output*) to every port of an agent. Fernandez extended Lafont's approach by defining an intersection type system for interaction nets [2]. It features type variables (supporting a form of polymorphism) and means to construct more complicated types (intersection types, arrow types). This system offers two features that may be useful to type monadic interaction rules: type variables and arrow types to denote functional types. However, it lacks abstract types with type variables, which are needed to specify the example data structures above. We are currently investigating how to extend Fernandez' type system with this feature.

The problem of type systems is connected to the problem of generic rules: Both deal with the

specification of interaction rules for a general range of agents. Obviously, type assignment to agents could prevent several cases of overlaps that may arise from the use of rules with generic agents. For example, just an assignment of polarity (as mentioned above) to the ports of every agent can eliminate overlaps between multiple generic rules. Further research is needed for a concrete model to support generic rules with an appropriate type system.

With regard to existing implementations, to the best of our knowledge no system based on interaction nets features the typing of agents and ports. Future work may include implementing such a type system in the programming language *inets* [6]. Currently, *inets* only supports *external datatypes*. These are values of a basic type (*int*, *char*, *float*,...) that can be attached to agents (for details, see [3]). However, the agents themselves (and their ports) do not have a type.

5 Conclusion

In this paper, we presented first steps of a new approach to handle side effects in interaction nets. This approach is based on monads, particularly on their usage in the functional language Haskell. We presented three different ad-hoc solutions for impure functions in interaction nets. We have shown that the monad laws hold for these solutions. This is an important step towards realizing impure functions. The presented monads act as a proof of concept. Furthermore, we have shown a more general result regarding the monad laws in interaction nets in Proposition 3.1.2. This result is tied to our modelling approach of the *return* and *bind* operators, which captures the essence of monads in a natural and intuitive way. Other modelling approaches using different agents/rules are currently under investigation.

Monads were originally defined in Category theory [11]. In this paper, we only considered the notion of a monad in the setting of functional programming. An approach that focuses more on category theory may offer a better level of abstraction to define a monad framework in interaction nets, and will be investigated as part of future work.

The similarity of the realization of the three monads is obvious. The same agents are used, and the rules work in a similar fashion. Our goal is to generalize these solutions to an abstract framework of handling side effects in interaction nets. In this paper, we discussed the two main obstacles towards such a framework: interaction rules with generic agents and a type system extension to appropriately specify monadic data structures and operators. Both theoretical results on and implementation of these features are subject of future work. Currently, we are working on a prototype implementation of generic interaction rules in the programming language *inets* [6] (including an underlying mechanism to perform I/O such as the *ccall* interface in [7]) and an extension of Fernandez' intersection type system for interaction nets [2].

Acknowledgements: We are grateful to the anonymous reviewers for their useful hints and suggestions.

5.1 Related Work

Extensions to interaction nets have been proposed in many directions. For example, *nested pattern matching*, an approach for allowing more complex rule patterns is developed in [4, 5]. Nested patterns relax the restriction of having only two agents in the left-hand side of a rule

while preserving the beneficial properties of interaction nets. Several monad rules in this paper make use of nested patterns. For example, rule (3) of the writer monad has a nested pattern.

Monads have been applied to various formalisms. A recent application outside of functional programming can be found in [8], where monads are used to structure mechanisms in interactive theorem provers. Several evaluators for interaction nets have been implemented, cf. for example [1, 12]. To the best of our knowledge, no system deals with impure functions in an appropriate way.

Bibliography

[1] Almeida, J.B., Pinto, J.S., Vilaca, M.: A tool for programming with interaction nets. Electronic Notes in Theoretical Computer Science 219, 83–96 (2008)

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

[3] Fernández, M., Mackie, I., Pinto, J.: Combining interaction nets with externally defined programs. In: Proc. Joint Conference on Declarative programming (APPIA-GULP-PRODE'01), Évora (2001), http://hdl.handle.net/1822/776

[4] Hassan, A., Jiresch, E., Sato, S.: An implementation of nested pattern matching in interaction nets. Electronic Notes in Theoretical Computer Science 21, 13–25 (2010)

[5] Hassan, A., Sato, S.: Interaction nets with nested pattern matching. Electr. Notes Theor. Comput. Sci. 203(1), 79–92 (2008)

[6] The inets project. http://www.interaction-nets.org

[7] Jones, S.P., Wadler, P.: Imperative functional programming. ACM Symposium on Principles of Programming Languages (POPL'02) (Oct 1992)

[8] Kircher, F., Munoz, C.: The proof monad. Journal of Logic and Algebraic Programming 79, 264–277 (2010)

[9] Lafont, Y.: Interaction nets. Proceedings, 17th ACM Symposium on Principles of Programming Languages (POPL'90) pp. 95–108 (1990)

[10] Mackie, I.: YALE: yet another lambda evaluator based on interaction nets. Internation Conference on Functional Programming (ICFP'98) pp. 117–128 (1998)

[11] Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1191)

[12] Pinto, J.S.: Parallel evaluation of interaction nets with mpine. In: Middeldorp, A. (ed.) Rewriting Techniques and Applications (RTA'01). Lecture Notes in Computer Science, vol. 2051, pp. 353–356. Springer (2001)

[13] Wadler, P.: How to declare an imperative. ACM Comp. Surveys 29(3), 240–263 (Sep 1997)