Realizing Monads in Interaction Nets via Generic Typed Rules

Eugen Jiresch and Bernhard Gramlich

{jiresch,gramlich}@logic.at

June 2011

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

TECHNISCHE UNIVERSITÄT WIEN

Technical Report E1852-2011-01

Theory and Logic Group, Institute of Computer Languages (E185/2)
TU Wien, Favoritenstraße 9, A-1040 Wien, Austria

Realizing Monads in Interaction Nets via Generic Typed Rules

Eugen Jiresch and Bernhard Gramlich[*]
Theory and Logic Group, Institute of Computer Languages
Faculty of Informatics, Vienna University of Technology
Favoritenstr. 9 – E185/2, A-1040 Wien, Austria

June 2011

Abstract

Interaction net systems are a model of computation based on graph rewriting. They enjoy various nice properties which make them a promising basis for a functional programming language. However, mechanisms to model *impure functions* are indispensable for a practical language. A natural approach to achieve this goal is the systematic use of *monads*. Yet, specifying the appropriate monads for impure language features is hard, due to the very restricted form of basic interaction rules. What is missing in particular, are appropriate means to specify higher-order functions and some typing mechanism that restricts computations to reasonable settings.

In this paper, we propose two extensions of interaction nets which solve these problems. First we extend interaction rules with *generic rules*, thus adding a form of higher-order functions. Moreover, we define constraints on these rules to ensure the preservation of uniform confluence. In addition, we propose a simple type system in order to appropriately restrict the matching of generic rules. Finally, we show how the combination of these features, i.e., generic typed rules, can indeed be employed to model impure functions in interaction nets via monads in an intuitive and simple manner.

1 Introduction

1.1 Background

One of the major challenges for any functional programming language is how it deals with computational side effects and impure functions. It may simply allow functions with side effects and thus restrict the possibility for equational reasoning on programs. ML is an example for this design decision. On the other hand, several techniques exist to model impure functions in a pure environment: examples are linear logic, continuations and monads. Haskell's monad framework has been very successful in modeling various forms of side effects while preserving the possibility to perform equational reasoning on programs.

Interaction nets are a new programming paradigm based on graph rewriting. The idea behind interaction nets is to represent programs as graphs (nets). Their execution is modeled by rewriting the graph based on specific node (agent) replacement rules. This simple

[*]Email: {jiresch,gramlich}@logic.at. The first author was supported by the Austrian Academy of Sciences (ÖAW) under grant no. 22932 and by a grant of the *Vienna PhD School of Informatics*

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 uniform 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.

1.2 Our Approach and Contribution

Interaction nets can be considered a pure, side effect free language. Our goal is to provide an extensible framework for interaction nets that handles various side effects such as I/O, exception handling or state manipulation. In [23], Mackie suggests the adaptation of monads to interaction nets to solve this problem. We have previously made initial progress in this direction: in [14], we defined a set of interaction rules that corresponds to the *Maybe* monad and showed that the monad laws hold. However, this encoding was only an ad-hoc solution, in the sense that it was specific for every individual monad rather than generic. This deficiency corresponds to the fact that monads are based on higher-order functions and abstract datatypes, which are not supported by interaction nets. The restricted form of interaction rules does not allow for a sufficiently general definition of the monadic operators. For example, consider Haskell's definition of the `>>=` (`bind`) operator of the *Maybe* monad:

```
>>= :: Maybe a -> (a -> Maybe b) -> Maybe b
(Just x) >>= f = f x
Nothing  >>= f = Nothing
```

The second argument of `bind` is an arbitrary function `f` (of type `a -> Maybe b`). Interaction rule patterns only consist of concrete agents: there is no equivalent of a function variable.

In this paper, we attempt to remove this deficiency by introducing *generic agents and rules*. Essentially, a generic rule pattern is a pair of one concrete agent and one generic agent that represents an arbitrary agent (similar to `f`'s role in the definition of `bind`). Such rules have already appeared in previous papers on interaction nets, but usually only in the form of duplication and deletion agents. However, we are not aware of any result on the preservation of uniform confluence in the presence of generic rules.

We define appropriate priority and well-definedness constraints on (applying) generic rules in order to preserve uniform confluence (of the induced reduction relation), thus yielding an extension of interaction net systems that still satisfies the uniform confluence property. In addition, we define *rule types* based on *port types*. This simple type system allows us to further restrict the matching of generic rules. We show how to use both generic rules and rule types to model monads in interaction nets in a suitable way.

Our contributions can be summarized as follows:

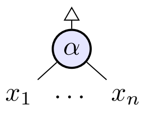
- A formal definition of generic rules, introducing a form of higher-order functions and partial evaluation (currying) to interaction nets.
- Appropriate constraints on the usage of generic rules in order to preserve uniform confluence.
- A simple type system to suitably restrict rule matching in the new setting.
- Two typical side effect examples where we model the corresponding monads with the above extensions of interaction nets.

In Section 2 we give a short introduction to interaction nets and discuss side effects and monads to motivate generic rules. Section 3 defines generic rules and constraints, including proofs of the preservation of uniform confluence. In Section 4, we define rule types and show how they can be used to restrict matching. We then give two examples of application of these extensions in Section 5. Finally, we conclude and discuss related work in 6.2.

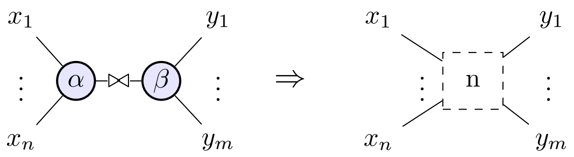
2 Preliminaries

2.1 Interaction Nets

Interaction nets (INs) have been introduced in [19]. A *net* is a graph consisting of *agents* (nodes) and *ports* (edges). Every agent has a label and an arity, denoting the number of ports that are connected to it. The agent α below is of arity n and has $n+1$ ports:



Computation is modeled by rewriting the graph, which is based on *interaction rules.* These are rules that specify how a subnet of an IN consisting of two nodes which are connected via their *principal ports* (denoted by the arrow) are rewritten. We refer to these two connected nodes as *active pair* or *redex.* Interaction rules preserve the *interface* of the net: no auxiliary ports are added or removed.



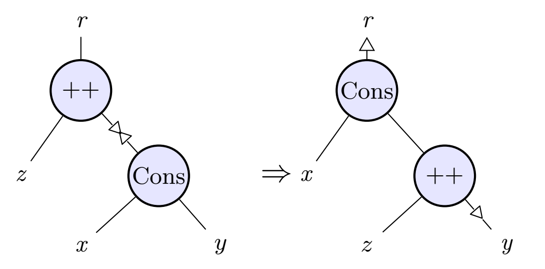
We will represent interaction rules textually as $A \bowtie B \Rightarrow N$, where $A \bowtie B$ represents the active pair on the left-hand side (LHS), and N the net on the right-hand side (RHS). If the RHS is not of importance for an argument, we may, just write $A \bowtie B$. We write $A \sim B$ to denote a specific active pair/redex (which is part of some net).[1] A net containing $A \sim B$ as a subnet can be rewritten by a rule $A \bowtie B$.

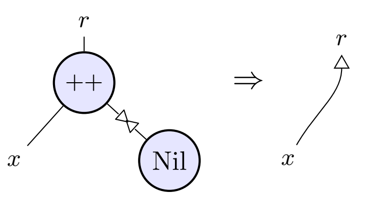
Definition 2.1.1 (interaction net system (INS)) *An* interaction net system (INS) *is a pair $\mathcal{R} = (\Sigma, R)$ where Σ is a signature, i.e., a set of agents, and R is a set of interaction rules, where for every rule $A \bowtie B \Rightarrow N \in R$ we have $A, B \in \Sigma$ and $A \neq B$. Moreover, there is at most one rule for every such active pair.*

Slightly abusing notation, we will often write R instead of $\mathcal{R}$, thus leaving the signature implicit.

Example 2.1.2 *Consider an INS that allows us to concatenate lists of natural numbers: $\mathcal{R} = (\Sigma, R)$ where $\Sigma = \{0^0, S^1, Cons^2, Nil^0, {+\!+}^2\}$ and R is defined as follows:*

[1]Note that both notations $A \bowtie B$ and $A \sim B$ are interpreted modulo commutativity.

(++) 



These rules correspond to the following function definition (in a Haskell-like syntax):

```
(++) :: [a] -> [a] -> [a]
(++) (Cons x y) z = Cons x ((++) y z)
(++) Nil x = x
```

Definition 2.1.3 (reduction relation) *Let (Σ, R) be an INS. The reduction relation $\Rightarrow_R$ induced by this system is defined as follows: $N \Rightarrow_R M$ if an active pair $A \sim B$ is a subnet of N, $(A \bowtie B \Rightarrow P) \in R$ and M can be obtained from N by replacing $A \sim B$ with P.*

If the set of rules is clear from the context, we simply write $\Rightarrow$ instead of $\Rightarrow_R$.

Proposition 2.1.4 (uniform confluence, Lafont [19]) *Let N be an interaction net. If $N \Rightarrow P$ and $N \Rightarrow Q$ where $P \neq Q$, then there exists a net R such that $P \Rightarrow R \Leftarrow Q$.* [2]

In [19], Lafont shows that three properties of interaction nets are sufficient for uniform confluence:

1) Linearity: Ports cannot be erased or duplicated via interaction rules.

2) Binary interaction: Agents can only be rewritten if they form an active pair, i.e., if they are connected via their principal ports.

3) No ambiguity: For each pair S, T, of distinct agents there is *at most one* interaction rule that can rewrite $S \sim T$, and there is no rule with LHS of shape $A \bowtie A$.

Uniform confluence ensures that the order of performing multiple computational steps does not affect the final result of the computation. As there is no interference between reductions, interaction rules may even be applied in parallel. This makes interaction nets a promising basis for a concurrent programming language. In addition, active pairs cannot be duplicated: This ensures that they are evaluated only once, which allows for *sharing of computation.* Only nets without active pairs (i.e., normal forms) can be duplicated.

Note that the restriction that no LHS should be of the form $A \bowtie A$ (i.e., *self-interaction*) can be weakened: In [20], Lafont suggests that an agent may interact with itself if the rule's

[2]Several publications on interaction nets, including [19], refer to this property as *strong confluence.* We use the term *uniform confluence (WCR^1)* in order to account for the fact that if P and Q are distinct, then one step is taken from either net to reach a common reduct.

RHS enjoys a certain natural form of symmetry (which guarantees that self-overlaps are always trivial, i.e., produce the same reduct). All results of the current paper can easily be transferred to this setting. In particular, any generic rule that potentially allows *self-interaction* would have to satisfy the mentioned symmetry constraint.

2.2 Side Effects, Monads and Generic Rules

Computational side effects like I/O and exception handling are crucial features of any practical programming language. Our goal is to promote interaction nets towards such a language: We are working on a general framework to represent side effects - also known as *impure functions* - in interaction nets. INSs can be considered a *pure* (functional) programming paradigm. The nice properties of INSs depend on this purity. As soon as interaction rules incorporate side effects, uniform confluence (and, hence, parallel evaluation) is generally lost.

Yet, it is well-known that computational side effects can be simulated by pure functions with additional arguments and return values. For example, consider the following function:

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

Obviously, `square` is an impure function: Besides computing the square of a given number, `square` accesses and increments a global variable `y`. However, this behavior can be modeled by providing `y` as a second argument to the function and returning its increment:

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

The function `square_inc` is free of side effects. It does not change the state of the machine or program, nor is its result affected by the former. The data dependency induced by the extra parameters results in a fixed order of evaluation. Manually adding such additional values and propagating them through of a program is of course tedious and error-prone, especially in the presence of multiple side effects. A more general solution is needed to provide a clean way of expressing impure functions.

Therefore, we decided to adapt an existing solution for our purposes: Monads have been used in *Haskell* with great success to model all kinds of impure functions. Originally being a notion from category theory, monads were adapted to handle side effects in functional programs, cf. e.g. [25, 27, 17, 28].

Essentially, a monad provides data structures and functions to model a side effect and its propagation with pure functions. More formally, a monad is a triple of an abstract datatype `M a` and two (higher-order) functions operating on this type:

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

The idea behind this triple is 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.

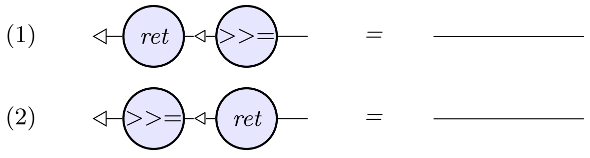
A monad needs to satisfy the following laws:

```
(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 equal 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`. Finally, `bind` has a property that is similar to associativity. This is expressed by law (3).

Monads are a promising candidate for realizing computational side effects in interaction nets. In [14], we have made a first step towards the adaptation of monads to interaction nets. There we defined concrete example monads for different side effects in interaction nets and showed that they satisfy the monad laws. However, the example monads only represented an ad-hoc solution: Basic interaction rules cannot express the higher-order character and complex type information of monadic data structures and functions. Consequently, our solution in [14] was specific for the concrete data structures involved, but not general in the sense of being parameterized (or *generic*). The goal of the paper is to extend interaction rules to allow for a more general, natural and adequate definition of monads. We define the notion of a monad in interaction nets as follows:

Definition 2.2.1 (Interaction Net Monad) *An* interaction net monad *is an INS that contains two unary agents* >>= *and* ret. *The rules of the INS need to satisfy the following equalities:*



Here, equality *is interpreted as* observational equivalence *(operationally expressed as convertibility, which, under confluence, is equivalent to joinability): If arbitrary nets are connected to the free ports of both nets of an equation (enabling reduction), then they can be reduced to a common successor.*

The equivalences (1) and (2) correspond to the first two monad laws. As we have shown in [15], the third monad law automatically holds due to the representation of the >>= and *ret* agents.

Our notion of equivalence is of course similar to other work on observational equivalence for interaction nets, such as [7]. These equivalences can be shown by giving reduction sequences of nets that yield the aforementioned common successors.

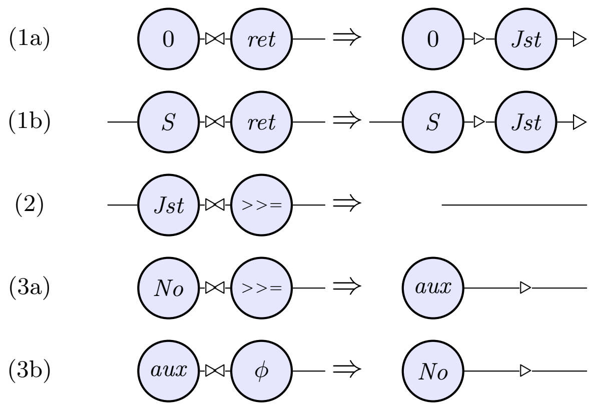
To illustrate the benefits of our approach to be developed, we will use the *Maybe* monad as a running example.

Example 2.2.2 *The Maybe monad is used in Haskell to model exception handling. It is defined as follows:*

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

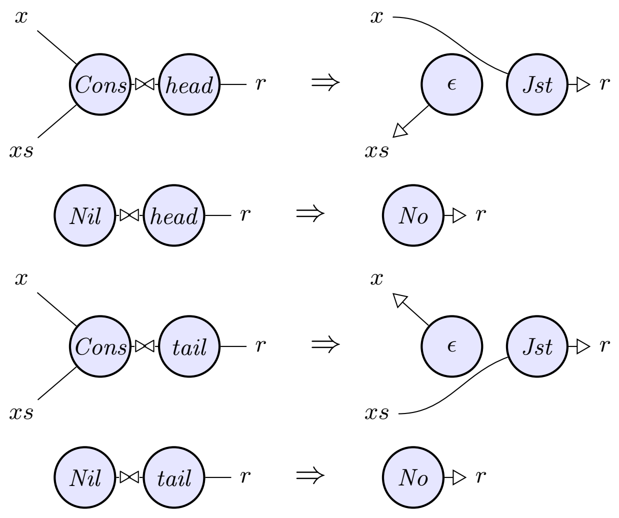
Values of the `Maybe` *data type are either a plain value (*`Just a`*) or the error value* `Nothing` *denoting an exception. Plain values are simply forwarded to a function f by* `bind`*, whereas* `Nothing` *is returned by* `bind` *(when given f as second argument) without using f at all.*

As shown in [14], the Maybe monad for natural numbers can be modeled in interaction nets:



The rule labels correspond to the textual definition of the monadic functions. When comparing both definitions, two aspects are apparent: First, *ret* only interacts with S and 0, i.e., symbolic natural numbers. Haskell's `return` is defined for an arbitrary data type `a`. Second, rule (3b) features a so far unmentioned agent ϕ. The symbol ϕ corresponds to `f` in the textual definition and represents an arbitrary agent. Analogously, `f` represents an arbitrary function. This interaction rule is special: Whereas the LHS of ordinary interaction rules consists of two specific agents, here we have a rule with an arbitrary agent in its LHS. Such *generic* rules and their properties are one of the main topics of this paper. We will discuss them in detail in the next section.

To correspond to Haskell's definition of the `Maybe` monad, ϕ should be any agent that interacts with an agent representing a plain value and returns a *Maybe* value via its auxiliary port. For example, consider the following interaction rules that model the *head* and *tail* operators on lists. Both agents return an exception value when interacting with an empty list.[3]



The agent ϵ in the RHS of the third rule erases any other agent. This is another case of an agent interacting with any arbitrary agent.

[3]We are aware of the fact that in Haskell *head* and *tail* do not return a *Maybe* value, but terminate with a (fatal) error when being called with an empty list. For the sake of simplicity we chose to make these functions "exception safe" here.

3 Generic Rules and Imposed Constraints

In this section, we introduce the generic rule extension for interaction nets. We define constraints on the application of these rules and show that these constraints are sufficient to preserve uniform confluence. In a nutshell, we impose that ordinary rules always have priority over generic rules and that multiple generic rules may not overlap.

An ordinary interaction rule matches one specific pair of (distinct) agents only. However, several papers ([4, 5, 21]) on interaction nets feature agents that interact with any (arbitrary) agent. Informally, these variable or *generic* agents assume the role of *functional variables*. We will refer to such rules as *generic rules*. Typical examples are the agents δ and ϵ, which duplicate and erase, respectively, any other agent.

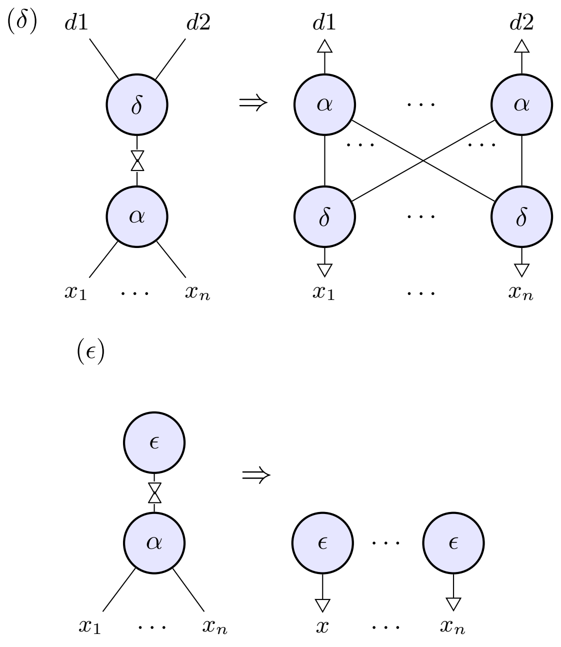


Figure 1: δ, ϵ: generic rules for duplication/deletion

In addition, exceptions to these rules can be given: An agent may interact differently with one specific agent than with all others. Usually, these rules are simply stated without a more detailed discussion of properties. However, it is obvious that such rules have the potential to destroy the uniform confluence property: In a system with *generic* rules, a given active pair may be matched by more than one rule. In fact, this is the case for any system that contains the δ and ϵ rules: the active pair $(\delta \sim \epsilon)$ can be reduced using either rule. This violates the *no ambiguity* property mentioned in Section 2.1 and generally destroys uniform confluence (although in this case, it does not: both rules yield the same net). Therefore, it is important to give a formal definition of how generic rules in INS are to be interpreted and restricted in such a way that uniform confluence is preserved.

Generic agents can be classified into two kinds, as shown in Figure 2. They may either have a fixed arity (such as ϕ in the *Maybe* monad example) or have a non-fixed, arbitrary number of auxiliary ports (such as δ or ϵ). We call the latter *variadic* or arbitrary-arity agents. Later in this section, we will extend variadic agents by *non-uniform* variadic agents, which — among their arbitrarily many ports — have a (fixed) number of ports that are handled in a specific way. Note that generic agents may only appear in interaction rules, not in concrete instances of nets.

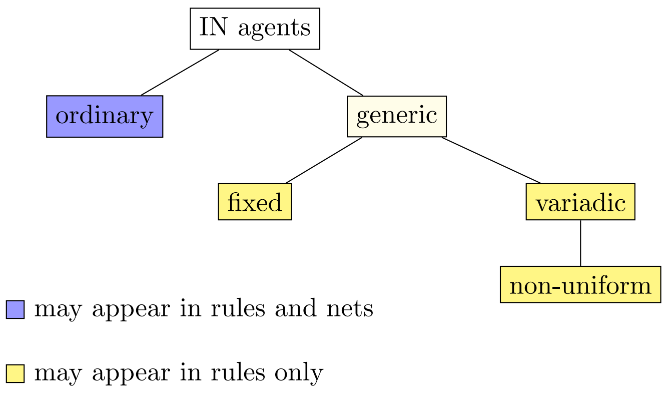


Figure 2: Classification of IN agents

3.1 Generic Rules

We now give a formal definition of (reduction with) generic rules and overlaps. Then we define two natural constraints on interaction net systems with generic rules.

Notation: We use upper-case letters (A,B,..) to denote specific (i.e., arbitrary, but fixed) agents, and lower case greek letters ($\alpha, \beta, \phi, \psi, \ldots$) to denote *generic* agents. We use INS_G for INS with generic rules.

Definition 3.1.1 (generic rules) *A* generic interaction rule *is an interaction rule $\alpha \bowtie B \Rightarrow N$ whose LHS consists of one generic agent α and one ordinary agent B. The RHS N may contain one or more occurrences of α.*

Definition 3.1.2 (rule matching) *An* ordinary *rule $A \bowtie B$ is* applicable *to (or* matches*) an active pair if the latter is of the shape $A \sim B$. A* generic *rule $\alpha \bowtie B$ is* applicable *to (or* matches*) an active pair if the latter is of shape $A \sim B$ (with $A \neq B$) and if α and A have the same number of ports. In this case, we also say that α* matches *A.*

Note that active pairs are not ordered: $A \sim B$ is equivalent to $B \sim A$, i.e., both are matched by a rule $A \bowtie B$.

In the presence of generic rules, more than one rule may match a given active pair. We now classify these overlaps according to the rules involved. Note that we are only interested in overlaps on the level of a single active pair/redex: Due to the *binary interaction* property, the reduction of one active pair cannot influence another one (e.g., erase or duplicate it). Therefore, we define overlaps as the matching of more than one rule on a single active pair.

Definition 3.1.3 (overlaps) *Two (distinct) rules in an INS_G* overlap *if there exists a single active pair (w.r.t. some INS) which is matched by both rules*[4] *and can also be rewritten by them.*[5]

Let (Σ, R) be an INS_G. Let $O(R) \subseteq R$ be the ordinary rules of R and $G(R) \subseteq R$ be the generic rules of R.

We say that two rules $A \bowtie B$ and $\alpha \bowtie B$ in an INS_G form an ordinary-generic-overlap *(* OG-overlap *for short) if both match an active pair $A \sim B$.*

Two generic rules $\alpha \bowtie B$ and $A \bowtie \beta$ form a generic-generic-overlap *(*GG-overlap *for short) if α matches A and β matches B, i.e., both rules match the active pair $A \sim B$.*

[4]Note that in *term rewriting* such kinds of special *overlaps* are also called *overlays*. In our setting, due to the non-nested patterns of LHSs in interaction net rules, overlaps cannot be "partial" or "properly inside" an LHS.

[5]This requirement ensures that reducing subnets in different ways by overlapping rules is indeed possible. Later on we will prevent such overlaps by restricting the reduction relation.

We now define our constraints for generic rules, first for agents with fixed arity, such as rule (3b) of the *Maybe* monad. Afterwards, we extend these constraints to generic agents with arbitrary arity (e.g., δ and ϵ).

3.2 The Default Priority Constraint (DPC)

The idea behind this constraint is to prevent OG-overlaps by giving priority to ordinary rules. If for an active pair an INS$_\mathrm{G}$ has no matching ordinary rule but a matching generic rule, then (and only in this case) the generic rule may be applied.

Definition 3.2.1 (default priority constraint (DPC)) *Let $\mathcal{R} = (\Sigma, R)$ be an INS$_\mathrm{G}$. Then $\mathcal{R}$ satisfies the* D*efault* P*riority* C*onstraint (DPC) if the induced reduction relation is restricted as follows: A generic rule $\alpha \bowtie B \in R$ is only applicable to an active pair $A \sim B$ if $A \bowtie B$ is not the LHS of any rule in R.*

In this case we write $\Rightarrow_{R_{DPC}}$ for the restricted reduction relation.

The DPC ensures that "exceptions" to generic rules assume priority. We now show that DPC completely prevents OG-overlaps. Our main argument is that adding a generic rule $\alpha \bowtie B \Rightarrow N \in R$ to an INS (Σ, R) is equivalent to adding an ordinary version $A \bowtie B \Rightarrow N[\alpha/A]$ of the rule for each symbol A in Σ (distinct from B). From now on we assume that Σ is finite.

Definition 3.2.2 (agent substitution) *Let N be an interaction net. An* agent substitution *is a mapping $\phi = \{\alpha_1/A_1, \ldots, \alpha_n/A_n\}$ with $arity(\alpha_i) = arity(A_i)$ whose application $\phi(N)$ to an interaction net N yields $N[\alpha_1/A_1, \ldots, \alpha_n/A_n]$, i.e., the interaction net N where every subnet α_i has been replaced by A_i.*

Definition 3.2.3 (unfolding) *Let $\mathcal{R}= (\Sigma, R)$ be an INS$_\mathrm{G}$. We define its* unfolding *$U(R)$ as follows: $U(R) = O(R) \cup \{A \bowtie B \Rightarrow N[\alpha/A] \mid (\alpha \bowtie B \Rightarrow N) \in G(R), A \in \Sigma, A \neq B, \alpha$ matches $A\}$.*

Proposition 3.2.4 *Let N, M be two INs in an INS$_\mathrm{G}\mathcal{R}$. Then $N \Rightarrow_R M$ if and only if $N \Rightarrow_{U(R)} M$.*
Proof: By complete case distinction.
$\Rightarrow$: Let $N \Rightarrow_R M$.
Case 1: An ordinary rule was used to reduce N to M. Then clearly $N \Rightarrow_{U(R)} M$.
Case 2: A generic rule $(\alpha \bowtie B \Rightarrow P)$ was used to reduce N to M. Let $A \sim B$ be the active pair in N that was reduced. Obviously, α matches A. Then, by the definition of Unfolding, $(A \bowtie B \Rightarrow P[\alpha/A]) \in U(R)$. Hence, $N \Rightarrow_{U(R)} M$.
$\Leftarrow$: Let $N \Rightarrow_{U(R)} M$.
Let $(A \bowtie B \Rightarrow P) \in U(R)$ be the rule used to reduce N to M. This means that either $(A \bowtie B \Rightarrow P) \in R$ or $(\alpha \bowtie B \Rightarrow P[A/\alpha]) \in R$, for some generic agent α. In both cases, $N \Rightarrow_R M$. ■

It is clear that the unfolding of R may violate the basic *no ambiguity* constraint of interaction nets. The unfolded set of rules could contain more than one rule with the same LHS. Obviously, this is the case if there is an OG-overlap between two rules of R.

Using Proposition 3.2.4, we can semantically express generic rules via their unfoldings. We can also define a version of unfolding that takes DPC into account. This unfolding does not add an ordinary rule if a rule with the same LHS already exists:

Definition 3.2.5 (DPC-unfolding) *Let $\mathcal{R}= (\Sigma, R)$ be an INS$_\mathrm{G}$. We define its* DPC-unfolding *$DU(R)$ as follows:*
$DU(R) = O(R) \cup \{A \bowtie B \Rightarrow N[\alpha/A] \mid (\alpha \bowtie B \Rightarrow N) \in G(R), \alpha$ matches $A, A \in \Sigma, A \neq B, (A \bowtie B \Rightarrow M) \notin R\}$.

Proposition 3.2.6 *Let N, M be two INS$_\mathrm{G}$s. $N \Rightarrow_{R_{DPC}} M$ if and only if $N \Rightarrow_{DU(R)} M$.*

Proof: By complete case distinction.
$\Rightarrow$: Suppose $N \Rightarrow_{R_{DPC}} M$. If this reduction was done by applying an ordinary rule, then clearly $N \Rightarrow_{DU(R)} M$. If a generic rule was applied, then no matching ordinary rule must have been in R. This means that the applied rule is an ordinary instance of the generic rule, hence it is in $DU(R)$. Thus, $N \Rightarrow_{DU(R)} M$.
$\Leftarrow$: Assume $N \Rightarrow_{DU(R)} M$. If this reduction was done by applying a rule that is ordinary in R, then $N \Rightarrow_{R_{DPC}} M$. If the rule is an instance of a generic rule in R, then by the definition of DPC Unfolding, there cannot be an ordinary rule in R matching the active pair that was reduced in $N \Rightarrow_{DU(R)} M$. Hence, $N \Rightarrow_{R_{DPC}} M$. ■

Using the equality of a system with generic rules and its unfolding, we can show that DPC prevents OG-overlaps.

Proposition 3.2.7 *Let $\mathcal{I} = (\Sigma, R)$ be an INS where $\Rightarrow_R$ has no GG-overlaps. Then, $\Rightarrow_{R_{DPC}}$ has no overlaps.*
Proof: Consider $DU(R)$. Clearly, $DU(R)$ only consists of ordinary rules and has no overlaps (i.e., there is at most one rule matching any active pair). Since $N \Rightarrow_{R_{DPC}} M$ is equivalent to $N \Rightarrow_{DU(R)} M$, $\Rightarrow_{R_{DPC}}$ has no overlaps. ■

3.3 The Generic Rule Constraint (GRC)

As the DPC prevents OG-overlaps, we now focus on preventing GG-overlaps, i.e., overlaps between multiple generic rules. Here, our approach is straightforward: We simply disallow overlapping generic rules or enforce a higher-priority ordinary rule.

Definition 3.3.1 (generic rule constraint) *Let $\mathcal{R} = (\Sigma, R)$ be an INS$_\text{G}$. R satisfies the* G*eneric* R*ule* C*onstraint (GRC) if for all $\alpha \bowtie B \Rightarrow N$, $A \bowtie \beta \Rightarrow M \in G(R)$, one of the following conditions holds:*

(1) α does not match A or β does not match B.

(2) $A \bowtie B \in O(R)$.

While DPC restricts the reduction relation, GRC is a constraint on the set of interaction rules. Condition (1) simply disallows any rules that may form a GG-overlap. Condition (2) requires an ordinary rule that matches the active pair which causes the overlap. This means that the combination of GRC and DPC prevents any rule overlaps.

Proposition 3.3.2 *Let $\mathcal{I} = (\Sigma, R)$ be an INS$_\text{G}$ that satisfies GRC. Then, $\Rightarrow_{R_{DPC}}$ satisfies the* no ambiguity *property.*
Proof: Clearly, $\Rightarrow_{R_{DPC}}$ prevents all OG-overlaps. This means that if there is an overlap, it must be a GG-overlap. We then make a case distinction based on the conditions satisfied by GRC:
Condition (1) holds: In this case, there are no GG-overlaps.
Condition (2) holds: There is a rule $A \bowtie B \in O(R)$ that matches the active pair causing a GG-overlap. This overlap is prevented in $\Rightarrow_{R_{DPC}}$, as the ordinary rule has priority over all generic ones. ■

Proposition 3.3.3 (uniform confluence) *Let $\mathcal{I} = (\Sigma, R)$ be an INS$_\text{G}$ that satisfies GRC. Then $\Rightarrow_{R_{DPC}}$ has the uniform confluence property.*
Proof: In Section 2.1, we defined three properties that are sufficient for uniform confluence. These properties are satisfied by INSs with ordinary rules only. The introduction of generic rules does not affect the *linearity* or *binary interaction* property. Furthermore, we have shown in Proposition 3.3.2 that $\Rightarrow_{R_{DPC}}$ satisfies the *no ambiguity* property by preventing all possible overlaps. Hence, $\Rightarrow_{R_{DPC}}$ satisfies uniform confluence. ■

3.4 Generic Rules with Variadic Agents

So far, our rule constraints only deal with generic agents with *fixed arity*. In this subsection, we extend our results to generic rules with *arbitrary arity*, or *variadic* agents. For example, we again refer to the δ and ϵ rules in Figure 1. These rules match any active pair that consists of δ/ϵ and an agent of arity between 0 and n (where n is considered the maximum arity of all agents in the signature).

We now extend rule matching for variadic agents. A variadic agent matches any ordinary agent.

Definition 3.4.1 (variadic rule matching) *Let $r = \alpha \bowtie B$ be a generic rule, where α is of arbitrary arity. Then, r matches an active pair $A \sim B$.*

Note that this definition of rule matching with a generic rule includes the degenerate case of 0 auxiliary ports.

As indicated in Figure 1 on page 4, rules with variadic agents may have an arbitrary number of identical agents (or subnets) in their RHS (denoted by "...". When such a rule $\alpha \bowtie B$ is applied to an active pair $A \sim B$, the pair is replaced by an adequate version of the RHS net, where the number of identical subnets is $arity(A)$. For a general precise specification of INS$_\mathrm{G}$s with variadic agents, an appropriate schema for specifying the rules and the induced reductions has to be defined. Due to lack of space we will not elaborate on that here.

The constraints and properties of fixed-arity generic rules can be extended to the arbitrary arity case. Again, we make use of the notion of *unfolding*.

Definition 3.4.2 (arity unfolding) *Let $\mathcal{I} = (\Sigma, R)$ be an INS. Let $O(R)$ be the set of ordinary rules, $G(R)$ the set of fixed-arity generic rules and $AG(R)$ the set of arbitrary-arity generic rules of R. Let $Ar(\Sigma)$ be the set of arities of all agents of Σ. We define the* arity unfolding *$AU(R)$ as follows: $AU(R) = O(R) \cup G(R) \cup \{A \bowtie \alpha_i \Rightarrow N[\alpha/\alpha_i] \mid (A \bowtie \alpha \Rightarrow N) \in AG(R), arity(\alpha_i) \in Ar(\Sigma)\}$.*

Informally, the arity unfolding adds a single fixed-arity generic rule for all possible arities of the generic agent (i.e., all arities of agents in Σ) in an arbitrary-arity generic rule. If Σ is finite, then $AU(R)$ has finitely many rules. Note that $N[\alpha/\alpha_i]$ is a RHS that contains $arity(\alpha_i)$ identical subnets (as mentioned above).

Proposition 3.4.3 *Let $\mathcal{I} = (\Sigma, R)$ be an INS$_\mathrm{G}$. The we have: $N \Rightarrow_R M$ iff $N \Rightarrow_{AU(R)} M$.*
Proof: By complete case distinction.
$\Rightarrow$: Let $N \Rightarrow_R M$.
Case 1: An ordinary or fixed-arity generic rule was used to reduce N to M. Then clearly $N \Rightarrow_{U(R)} M$.
Case 2: A generic rule $(\alpha \bowtie B \Rightarrow P)$ was used to reduce N to M, where α is variadic. Let $A \sim B$ be the active pair in N that was reduced. Since $AU(R)$ contains a rule $\alpha_i \bowtie B \Rightarrow P[\alpha/\alpha_i]$ s.t. α_i matches A, $N \Rightarrow_{U(R)} M$.
$\Leftarrow$: Let $N \Rightarrow_{AU(R)} M$.
Case 1: An ordinary or fixed-arity generic rule of R was used to reduce N to M. Then clearly $N \Rightarrow_R M$.
Case 2: Let $(\alpha_i \bowtie B \Rightarrow P) \in AU(R)$ be the rule used to reduce N to M, where $r = (\alpha \bowtie B \Rightarrow P) \in R$. Let $A \sim B$ the active pair that was reduced. By Definition 3.4.1, r matches $A \sim B$. Hence, $N \Rightarrow_R M$.
■

Theorem 3.4.4 *Let $\mathcal{I} = (\Sigma, R)$ be an INS$_\mathrm{G}$, where R contains at least one generic rule with a variadic agent. If $AU(R)$ satisfies GRC, then $\Rightarrow_{R_{DPC}}$ satisfies the* no ambiguity *property.*
Proof: If $AU(R)$ satisfies GRC, then it has no GG-overlaps, except those that are prevented by DPC. It follows from Proposition 3.4.3 that R has no additional GG-overlaps either. Hence, $\Rightarrow_{R_{DPC}}$ has no overlaps. ■

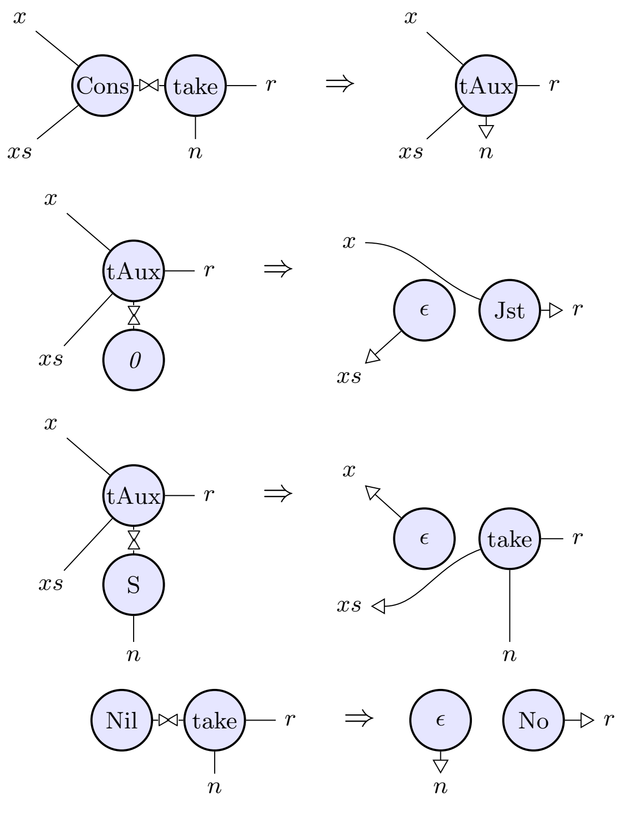
3.4.1 Non-Uniform Port Handling

The generic rules that feature variadic agents we have dealt with so far have one thing in common: In the RHS of the rule, all (arbitrarily many) ports of the generic agent are handled in the same, *uniform* way. However, we would like to define generic rules where a few selected ports of a generic agent receive a "special treatment". The remaining, arbitrarily many ports are handled uniformly. The motivation for this feature is simple: Generic rules should support the interaction nets equivalent of a *curried function*. For instance, in Haskell, the expression `(1+)` is a *partial application* of the addition function. It is a valid function that takes one argument and returns its increment by one. Partial application is an important feature of functional programming that we wish to capture with our generic rule extension. Using an example, we will see that *non-uniform port handling* can be used to achieve this in an intuitive fashion.

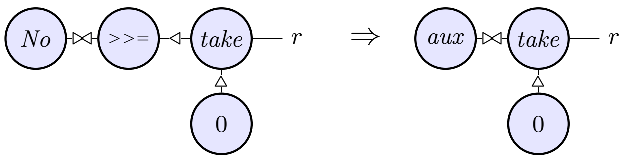
Example 3.4.5 *Consider an agent take of arity 2. Given a list and a number n, take returns the nth element of the list, or an exception if the list has less than n elements: Textually,* `take` *would be defined as:*

```
take :: Int -> [a] -> Maybe a
take 0 (Cons x xs) = Just x
take n (Cons x xs) = take (n - 1) xs
take n Nil = Nothing
```

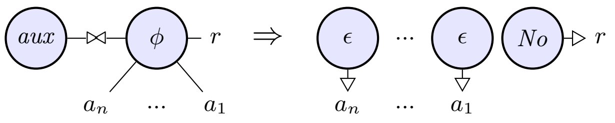
The following interaction rules define take*:*



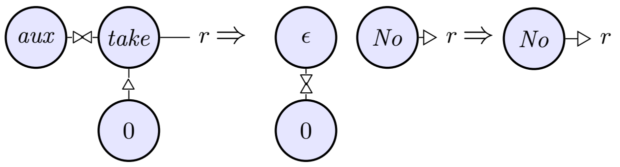
It is straightforward to see a net consisting of the agents *take* and 0 as the interaction net equivalent of the partially evaluated function `(take 0)`. Unfortunately, the interaction rules of the *Maybe* monad from Example 2.2.2 are not applicable here. Consider the following reduction:



Even though the second net has an active pair, it cannot be reduced any further. The *take* agent has two auxiliary ports and hence does not match the generic rule (3b). However, we can model the desired behavior of the *aux* agent with the following arbitrary-arity generic rule:



The port r corresponds to the output of *take* in the previous example. The updated generic rule is applicable to the net above, yielding the desired result:



Rule matching for the non-uniform case works almost identically to Definition 3.4.1. However, the respective agent of the active pair needs to have at least the number of non-uniformly handled auxiliary ports.

Definition 3.4.6 (non-uniform variadic rule matching) *Let $r = \alpha \bowtie B$ be a generic rule, where α is of arbitrary arity, but contains n auxiliary ports that are handled non-uniformly. Let $A \sim B$ be an active pair. r matches $A \sim B$ if $arity(A) \geq n$.*

Generic rule constraints and non-uniform port handling The generic rule constraints and their properties are almost the same for uniform and non-uniform variadic agents: in the latter case, the *arity unfolding* only creates fixed-arity generic agents (and rules) with arity greater or equal to the number of non-uniform ports. This is analogous to the arity restriction of the previous definition. It does not affect Theorem 3.4.4 and its result on uniform confluence.

4 A Simple Typing Approach

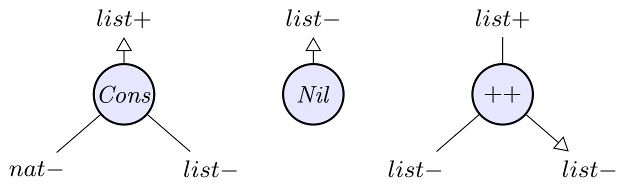
In the previous section, we introduced generic rules and suitable constraints on them to preserve uniform confluence. However, even if we satisfy these constraints, the matching capabilities of generic rules are too powerful. A generic rule would be applicable to more active pairs than intended. Consider the monadic function *bind* (>>=): It requires two arguments, one of which is a data type and one a function. The generic interaction rules for *bind* only require their "second argument" to have at least one auxiliary port. Therefore, *bind* may interact with any agent with an auxiliary port, whether it can be considered to represent a function or not. Moreover, this agent is supposed to return a monadic data type, which needs to be verified in some form.

Naturally, this problem is solved by a type system for interaction nets. We can restrict the type of an agent that may match the rule's LHS. In this section, we will define a type system that is suitable for expressing monadic rules in interaction nets.

4.1 Existing Typing Approaches

Several type systems have been proposed for interaction nets. In [19], Lafont defines a simple system that assigns a *base type* (int, char, . . .) to every port of an agent. Additionally, a *polarity* (+/−)

is assigned to each port. Agents may only be connected via ports of the same type, but opposite polarity. Intuitively, polarities divide ports into input and output ports. For example, the list and concatenation agents may be typed as follows:



This system is not expressive enough for generic rules and agents that model abstract datatypes (which are a part of monads). For example, this system does not feature type variables, which are essential for expressing monadic data types such as *Maybe a*.

In [6], Fernandez extends Lafont's type system by adding type variables and constructs for more complex types (arrows, intersections). While this *intersection type system* is more expressive, it is also more difficult to handle: Type assignment of intersection types is shown to be undecidable for interaction nets. Additionally, the procedure of generating all valid type instances is considerably complicated in the presence of intersections, requiring operations such as expansion or lifting in addition to (the usual) substitution.

4.2 Our Typing Approach

The main goal of our type system is to restrict the matching power of generic rules. We decided on a straightforward approach, keeping the system as simple as possible. However, our system needs to be expressive enough to model types of monadic functions, in particular *bind*:

```
>>= :: M a -> (a -> M b) -> M b
```

This means that a suitable type system needs to feature type variables, and a form of *arrow* (i.e., *functional*) *types*. While the intersection type system of [6] offers arrow types for ports, we feel that this system is overly complex for our needs. Moreover, the focus of intersection types is on criteria for termination of interaction net systems, while ours is to model matching of higher-order functions via generic rules. Therefore, we take the following approach: We restrict types for ports to base types and type variables. Base types may either be *type constants* (like *int*) or have *type parameters* (e.g., *list(int)*). More complex types are only defined implicitly via the set of all port types of an agent, also referred to as its *environment*. When matching a rule with an active pair, we compare the environment of the active pair's agents with the rule LHS.

Definition 4.2.1 (port types) *Let S be a set of* base types *(or* sorts*). Let each sort have an* arity*, denoting the number of type parameters. Let V be a set of* type variables *and $P = \{+, -\}$ be the set of* polarities*. The set of port types PT is defined as:*

- $v^p \in PT$, *where* $v \in V, p \in P$
- $s^p \in PT$, *where* $s \in S, p \in P$, *and* $arity(s) = 0$
- $s(t_1, \ldots, t_n)^p \in PT$, *where* $s \in S, p \in P, t_i \in PT (1 \leq i \leq n)$, *and* $arity(s) = n$

In the following examples, we use the greek letters τ, ρ to denote port type variables.

Considering only type constants (i.e., sorts with arity 0), a net is *well-typed* if all connections are between ports of different polarity, but with the same type. When adding type variables and parametrized types, we have to adapt this notion slightly:

Definition 4.2.2 (well-typed nets) *Given the a set of agents and the types of their ports, an interaction net N is* well-typed *if the following conditions hold:*

- *all connected ports are of opposite polarity.*

- *for all pairs of types of connected ports $(t_1, r_1), \ldots, (t_n, r_n)$, there is a solution to the unification problem $\{t_1 \approx r_1, \ldots, t_n \approx r_n\}$.*

Port types are the basic ingredients of our rule typing system. All types of an agent's ports form its *environment.* This notion was already introduced in [6]. However, we define it in a slightly different way, ordering the types by their port positions, starting counter-clockwise from the principal port.

Definition 4.2.3 (environment) *Let A be an agent of arity n. The* environment $\varepsilon(A)$ *is defined as $\{t_p, t_1, \ldots, t_n\}$, where t_p is the type of the principal port, and t_i is the type of the ith auxiliary port (viewed counter-clockwise from the principal port).*

For example, $\varepsilon(Cons) = \{list(\tau)^+, \tau^-, list(\tau)^-\}$ and $\varepsilon(++) = \{list(\tau)^-, list(\tau)^+, list(\tau)^-\}$.

Within the environment, multiple occurrences of the same variable (here: τ) refer to the same type. This is expressed by the *scope* of a type variable.

Definition 4.2.4 (variable scope) *The* scope *of a port type variable is defined as the agent (or environment) that the port is associated with.*

Based on the environment, we define a *rule type* that will be used for matching. In addition, we need a formalism to denote whether specific port type variables of the agents of a rule LHS correspond to each other. For this, we will use *type substitutions*:

Definition 4.2.5 (type substitution) *A* type substitution *is a substitution σ that maps port type variables to port types. A type substitution σ is represented by a set of shape $\{\tau_1 \mapsto \tau_1\sigma, \ldots, \tau_n \mapsto \tau_n\sigma\}$.*

Definition 4.2.6 (rule type) *Let $r = (A \bowtie B \Rightarrow N)$ be an interaction rule. Let the type variables of the ports of A and B be disjoint. The* rule type $RT(r)$ *is a triple $(\varepsilon(A), \varepsilon(B), S)$, where S is a* type substitution $\{\tau_1 \mapsto t_1, \ldots, \tau_n \mapsto t_n\}$ *consisting of types of either $\varepsilon(A)$ or $\varepsilon(B)$.*

The idea behind S is to ensure that the net of the rule is well-typed and specific interface ports of both agents share the same type.

Example 4.2.7 *Consider rule (2) of the Maybe monad in Example 2.2.2: Let $\varepsilon(Jst) = \{maybe(\tau)^+, \tau^-\}$ and $\varepsilon(>>=) = \{maybe(\rho)^-, \rho^-\}$, where maybe is a base type of arity 1 and τ, ρ are type variables. Then, $RT((2)) = (\varepsilon(Jst), \varepsilon(>>=), \{\tau \mapsto \rho\})$. It is important that both auxiliary ports share the same type: The auxiliary port of bind must be connected to an agent that matches the auxiliary port type of Jst.*

Note that we have not yet considered the type of the RHS of the rule. This is because we are mainly interested in matching. However, it is important that the types of the interface of the net are preserved during rule application. Therefore, we adapt the notion of well-typed rules from [19].

Definition 4.2.8 (well-typed rules) *A rule is well-typed if*

- *both nets of the LHS and RHS are well-typed.*
- *the types of all free ports are the same in the LHS and RHS.*

We say that a set of rules R is well-typed if all rules in R are well-typed. We will use the abbreviation INS_{GT} for an INS_{G} with typed rules.

Proposition 4.2.9 (subject reduction) *Let $\mathcal{R} = (\Sigma, R)$ be an INS_{G} where R is well-typed. Then, for two nets M, N such that $M \Rightarrow_R N$, the interface ports of M have the same type as the interface ports of N.*

Proof: Let $M \Rightarrow_R N$ by reduction of an active pair $A \sim B$ in M. We can distinguish two cases:
Case 1. No port of $A \sim B$ is part of the interface of M. Then, the interface of M is not affected by the reduction of $A \sim B$.
Case 2. One or more ports of $A \sim B$ are free, i.e., they are part of the interface of M. Then, it follows from the definition of a well-typed rule that the free ports of $A \sim B$ preserve their type during reduction. Hence, the interface ports of N have the same type(s) as the ones of M. ■

We now define matching of generic typed rules and active pairs, which is the main purpose of our type system. Informally, we match the environment of the generic agent and the corresponding agent of the active pair.

Definition 4.2.10 (typed rule matching) *Let $r = \alpha \bowtie B$ be a well-typed interaction rule where α is a generic agent. Let N a well-typed net containing an active pair $A \sim B$. We say that r matches $A \sim B$ if the following conditions hold:*

- *r matches $A \sim B$ w.r.t. Definition 3.1.2.*
- *there exists a type substitution σ s.t. $\sigma(S(\varepsilon(\alpha))) = \varepsilon(A))$, where S is the substitution of $RT(r)$.*

4.3 Typing for the Variadic Agent Case

So far, our type system can describe and match rules that consist of ordinary and fixed-arity generic rules. Since we want to model the environment of generic agents with arbitrary arity, we introduce a special symbol $*$ that may match any number of port types. This corresponds to the generic agents' capability to match agents with any number of ports of any type.

Definition 4.3.1 (type wildcard) *Let α be a generic agent that has an arbitrary number of auxiliary ports. We then model its environment as $\varepsilon(\alpha) = \{t_p, *\}$ where t_p is the type of the principal port of α and $*$ is called the* type wildcard*, corresponding to all auxiliary ports.*

Intuitively, $*$ may be replaced by any number of port types. In addition, generic agents and rules with non-uniform port handling can easily be expressed with this notation: For example, the generic agent ϕ in Section 3.4.1 can be modeled as $\varepsilon(\phi) = \{a^-, *, maybe(\alpha)^+\}$.

Typed rule matching can be extended with variadic agents as follows: when matching a rule with an active pair, we treat a variadic agent as a fixed arity agent: this agent has the arity of the corresponding agent of the active pair. The auxiliary ports covered by the type wildcard * are fresh type variables with suitable polarities.

Definition 4.3.2 (typed rule matching with variadic agents) *Let $r = \alpha \bowtie B$, where α is a variadic agent. Let $\varepsilon(\alpha) = (t^p, *, {t_1}^{q_1}, \ldots, {t_m}^{q_m})$, where t_i are the types of the non-uniformly handled ports (and q_i the respective polarities). $\alpha \bowtie B$ matches an active pair $A \sim B$ if the following holds:*

- *Let α' be a fixed-arity generic agent s.t. $\varepsilon(\alpha') = (t^p, {x_1}^{p_1}, \ldots, {x_n}^{p_n}, {t_1}^{q_1}, \ldots, {t_m}^{q_m})$, where $n = arity(B) - m$ (m is the number of non-uniformly handled ports), x_i are fresh variables and the polarities p_i are the polarities of B's auxiliary ports.*
- *$\alpha' \bowtie B$ matches $A \sim B$ w.r.t. Definition 4.2.10.*

Example 4.3.3 *Consider the take agent from Example 3.4.5 and the modified rule of the Maybe monad to support non-uniform port handling: The rule aux $\bowtie \phi$ is typed as $\{\{{\tau_1}^+\}, \{{\tau_2}^-, *, maybe(\rho)^+\}, \{\tau_1 \mapsto \tau_2\}\}$.*

Consider the active pair aux $\sim$ take, where $\varepsilon(take) = \{list(\tau)^-, int^-, maybe(\tau)^+\}$. By Definition 4.2.10, this pair is matched by the rule aux $\bowtie \phi'$, where $\varepsilon(\phi') = \{{\tau_2}^-, {\tau_3}^-, maybe(\rho)^+\}$.

4.4 Properties of the Type System

The presented type system combines well with the generic rule constraints of Section 3: The respective properties to preserve uniform confluence need not be changed in the typed setting. The type system only restricts the notion of matching used in the definition of the constraints. The constraints still imply the *no ambiguity* property.

To conclude this section, we show that well-typedness of nets is decidable in linear time, as it can be reduced to unification.

Proposition 4.4.1 *Let N be a typed net without generic agents. Given the environments of all agents, well-typedness of the net can be decided in linear time (i.e., linear in the number of agents and ports of the net).*
Proof: By Definition 4.2.2, two properties need to hold:

- all connected ports are of opposite polarity.
- for all pairs of types of connected ports $(t_1, r_1), \ldots, (t_n, r_n)$, there is a solution to the unification problem $\{t_1 \approx r_1, \ldots, t_n \approx r_n\}$.

The first property can be checked in linear time by traversing the net. The second property is a unification problem, which can be solved in linear time [2]. Therefore, the overall time complexity of deciding well-typedness of a net is linear to the number of agents and ports of the net. ■

5 Application: Monads in Interaction Nets

In this section, we give two examples of concrete interaction net systems that take full advantage of generic rules and rule types. We first present an improved version of the *Maybe* monad, and then give an INS$_{\mathrm{GT}}$that models the *Writer* monad (for logging/profiling).

Correctness of the Monads Both of the following interaction net monads are correct in the sense that they satisfy the monad laws mentioned in Section 2.2. This can be shown by a reduction of nets, such that both sides of each equation have a common reduct. In [14, 16], we gave proofs for the ad-hoc versions of both monads. The extensions of this paper do not affect the idea behind these proofs: generic rules essentially allow arbitrary agents as input for the monadic functions. In [16], the monad laws were already shown using reductions of arbitrary nets. Therefore, the proofs for the updated monad INSs are almost identical. Due to space constraints, we omit the full details.

5.1 The Maybe Monad Revisited

Using generic rules, we can express the interaction rules of the *Maybe* monad in a way that models Haskell's definition more closely. The INS$_{\mathrm{GT}}$*Maybe* is defined as $(\{Jst^1, No^1, ret^1, bind^1, aux^1\}, M)$ where M consists of the rules in Figure 3.

As we can see, the rules (1a) and (1b) have been merged to a single rule, using a variadic agent as the "argument" of *ret*. Rule (3b) now features a variadic with non-uniform port handling (as introduced in Section 3.4.1). To ensure that the set of rules satisfies the GRC, we introduce the auxiliary rule (GRC).

Proposition 5.1.1 $\Rightarrow_{M_{DPC}}$ *satisfies the uniform confluence property.*
Proof: The arity unfolding $AU(M)$ satisfies the GRC: There is a GG-overlap between the (unfolded) rules (1) and (3b). However, M contains the ordinary rule (GRC) that matches the active pair $(aux \sim ret)$. Therefore, the GRC is satisfied. By Theorem 3.4.4, $\Rightarrow_{M_{DPC}}$ satisfies the uniform confluence property. ■

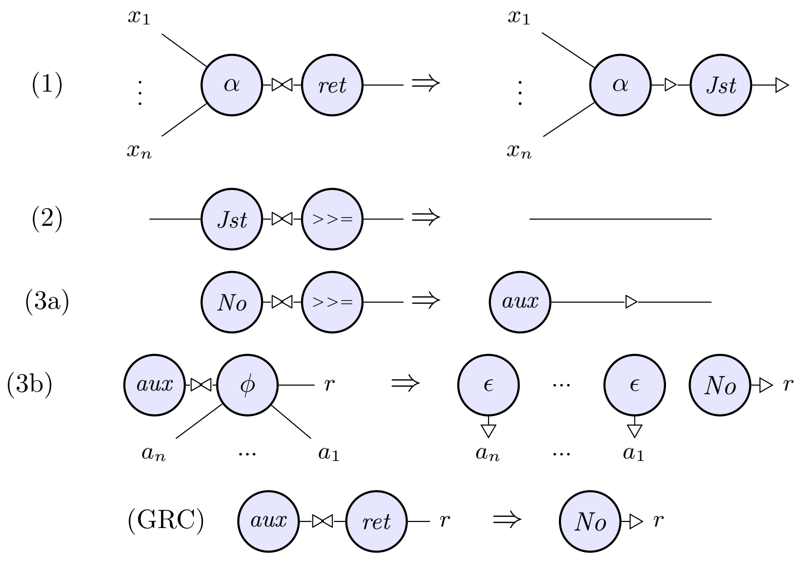


Figure 3: The rules of the *Maybe* monad

5.1.1 Port and Rule Types

We assign the following environment to the agents of the *Maybe* monad, where *maybe* is a sort of arity 1 and τ, ρ are type variables.

$$\begin{aligned}
\varepsilon(Jst) &= \{maybe(\tau)^+, \tau^-\} \\
\varepsilon(No) &= \{maybe(\tau)^+\} \\
\varepsilon(ret) &= \{\tau^-, maybe(\tau)^+\} \\
\varepsilon(>>=) &= \{maybe(\tau)^-, (\tau)^+\} \\
\varepsilon(au\tau) &= \{\ \tau^+\} \\
\\
\varepsilon(\alpha) &= \{\tau^+, *\} \\
\varepsilon(\phi) &= \{\tau^-, *, maybe(\rho)^+\}
\end{aligned}$$

The rule types are defined as follows. Recall that the scope of port type variables is the agent's environment. We add subscripts to the variables to denote which agent they belong to: τ_1 belongs to the first agent of the rule LHS and τ_2 to the second one.

$$\begin{aligned}
RT(1) &= \{\varepsilon(\alpha), \varepsilon(ret), \{\}\} \\
RT(2) &= \{\varepsilon(Jst), \varepsilon(>>=), \{\tau_1 \mapsto \tau_2\}\} \\
RT(3a) &= \{\varepsilon(No), \varepsilon(>>=), \{\}\} \\
RT(3b) &= \{\varepsilon(aux), \varepsilon(\phi), \{\}\} \\
RT(GRC) &= \{\varepsilon(aux), \varepsilon(ret), \{\}\}
\end{aligned}$$

5.2 The Writer Monad

The *Writer* Monad is used to add an optional, secondary output to a function and to collect these additional results of all functions in a log. Essentially, this can be modeled by maintaining a list of secondary results and appending the output of each function during evaluation. The `Writer` monad can be defined textually as:

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

`S` is the type of the secondary output. Together with a value `e` of type `S` and a binary function `':'` , `S` forms a *monoid*. This property is necessary to ensure the correctness of the `Writer` monad. The `Log` datatype is a pair of a values of types `a` and `S`. The operator `return` simply returns a pair of its argument and the identity element `e`. `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.

In interaction nets, we will model the log as a list where each element represents the secondary output of one function. The INS_{GT} *Writer* is defined as $(log^2, ret^1, bind^1, aux^2, ext^2\}, W)$, where W consists of the rules in Figure 4.

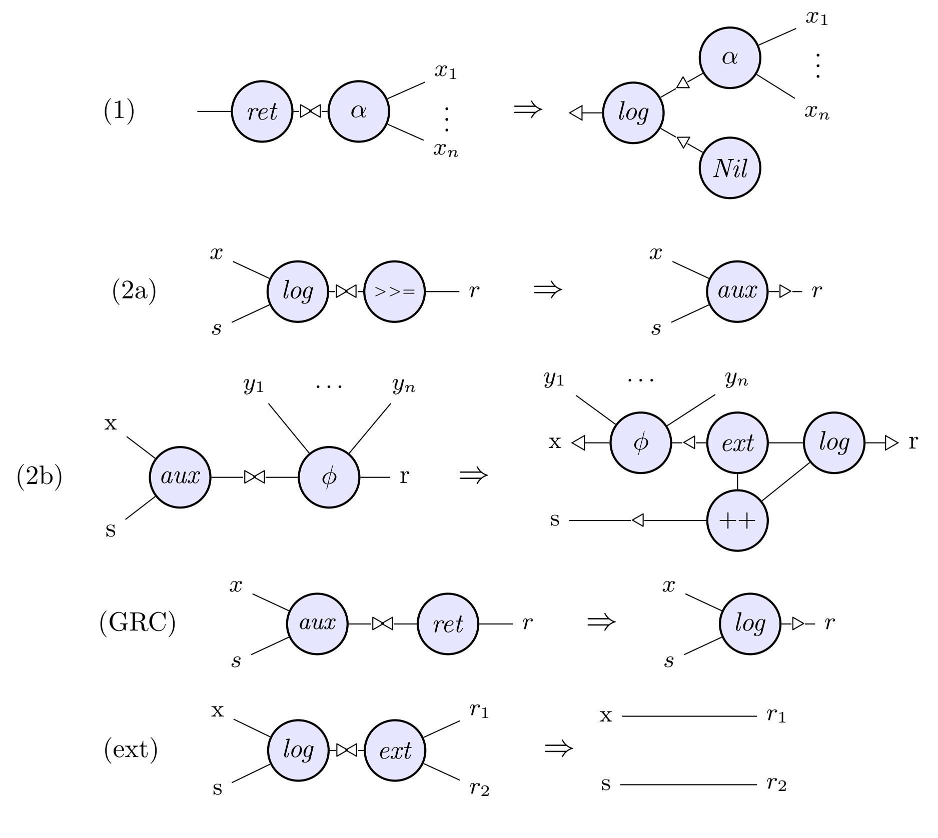


Figure 4: The rules of the *Writer* monad

Similarly to the *Maybe* monad, we can show that the *Writer* monad satisfies uniform confluence.

Proposition 5.2.1 *$\Rightarrow_{M_{DPC}}$ satisfies the uniform confluence property.*

Proof: The arity unfolding $AU(M)$ satisfies GRC: There is a GG-overlap between the (unfolded) rules (1) and (3b). Both rules match the active pair ($aux \sim ret$). Since M contains an ordinary rule with this active pair as LHS, M satisfies GRC. Hence, by Proposition 3.4.4 $\Rightarrow_{M_{DPC}}$ satisfies the uniform confluence property. ■

5.2.1 Port and Rule Types

The following environment types the agents of the *Writer* monad. In the interaction net setting, the *Log* datatype translates to an agent that has two distinct type variables, one for the primary return value of a function, and the other for the log value. We use the sorts log^2 and $list^1$ and the type variables τ, ρ.

$$\begin{aligned}
\varepsilon(log) &= \{log(\tau,\rho)^+, \tau^-, list(\rho)^-\}\\
\varepsilon(ret) &= \{\tau^-, log(\tau,\rho)^+\}\\
\varepsilon(>>=) &= \{log(\tau,\rho)^-, \tau^+\}\\
\varepsilon(aux) &= \{\tau^+, \tau^-, list(\rho)^+\}\\
\varepsilon(ext) &= \{log(\tau,\rho)^-, \tau^+, list(\rho)^+\}
\end{aligned}$$

$$\begin{aligned}
\varepsilon(\alpha) &= \{\tau^+, *\}\\
\varepsilon(\phi) &= \{\tau^-, log(\tau,\rho)^+, *\}
\end{aligned}$$

The *Writer* rules can be typed as follows. Again, we use subscripts to distinguish the type variables of both agents.

$$\begin{aligned}
RT(1) &= \{\varepsilon(ret), \varepsilon(\alpha), \{\}\}\\
RT(2a) &= \{\varepsilon(log), \varepsilon(>>=), \{\tau_1 \mapsto \tau_2\}\}\\
RT(2b) &= \{\varepsilon(aux), \varepsilon(\phi), \{\rho_1 \mapsto \rho_2\}\}\\
RT(GRC) &= \{\varepsilon(aux), \varepsilon(ret), \{\rho_1 \mapsto \rho_2\}\}\\
RT(ext) &= \{\varepsilon(log), \varepsilon(ext), \{\tau_1 \mapsto \tau_2, \rho_1 \mapsto \rho_2\}\}
\end{aligned}$$

5.3 The State Transformer Monad

The *State Transformer* monad is another well-known monad in Haskell. It models the application of a program w.r.t. a mutable state. It is defined as follows:

```
        data State s a  =  s -> (a,s)
(1)     return x  =  \s -> (x,s)
(2)     m >>= f  =  \r -> (\let (x,s) = m r in (f x) s)
```

A state transformer is a function that takes a state `s` as input and returns a pair of a return value `t` and a possibly changed state. In (1), `return x` is the function that returns a value `x` and the unchanged input state. In (2), `m >>= f` first applies the state transformer `m` to the input state and then `f` to the value and state of the resulting pair.

When realizing this monad as an INS, the main difference to the *Maybe* Monad lies in the monadic datatype `State a s`, which is a function itself. To account for this, we use agents for lambda terms and function application to "encapsulate" the state transformer function. Such agents have already been used in several papers (e.g., [24, 21]). Here, the λ agent acts as a constructor for `State` function objects. The application agent @ takes a role similar to Haskell's `runState` function. The drawback of this approach is that the INS is complicated with additional agents and rules.[6]

[6]We conjecture that the *rule archetype* of [24] can be used to improve this – cf. Section 6.

The *State Transformer* INS, including rules for explicit function application and handling of pairs, is shown in Figure 5.

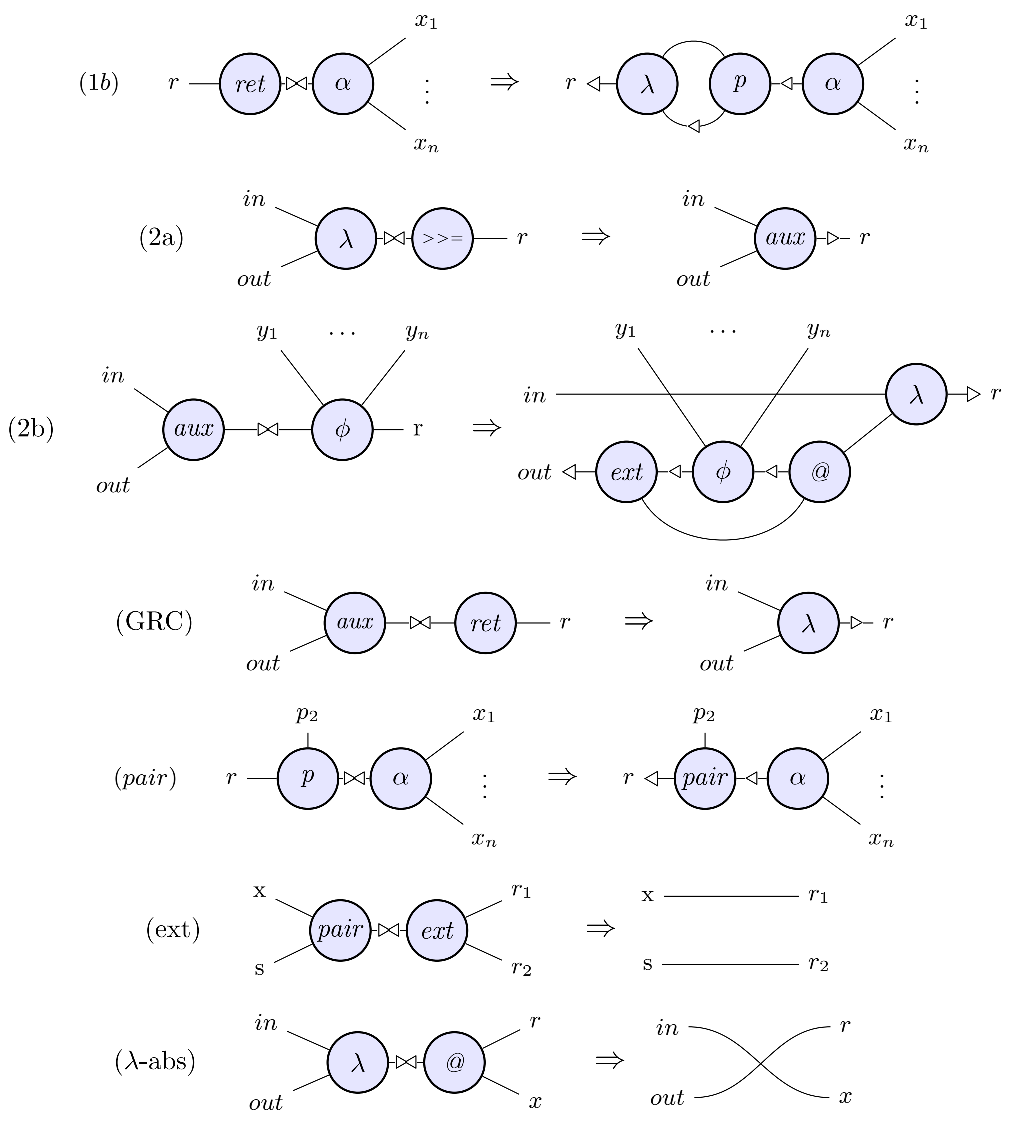


Figure 5: The rules of the *State Transformer* monad

Port and Rule Types We type the *State Transformer* Monad with the following environment. We use the sorts $state^2$ and $pair^2$ and the type variables τ, ρ.

$$
\begin{aligned}
\varepsilon(\lambda) &= \{state(\tau,\rho)^+, \tau^+, pair(\tau,\rho)^-\}\\
\varepsilon(ret) &= \{\tau^-, state(\tau,\rho)^+\}\\
\varepsilon(>>=) &= \{state(\tau,\rho)^-, \tau^+\}\\
\varepsilon(aux) &= \{\tau^+, \tau^+, pair(tau,\rho)^-\}\\
\varepsilon(ext) &= \{pair(\tau,\rho)^-, \tau^+, \rho^+\}\\
\varepsilon(pair) &= \{pair(\tau,\rho)^+, \tau^-, \rho^-\}\\
\varepsilon(p) &= \{\rho^-, pair(\tau,\rho)^+, \tau^-\}\\
\varepsilon(@) &= \{state(\tau,\rho)^-, \tau^-, pair(\tau,\rho)^+\}
\end{aligned}
$$

$$
\begin{aligned}
\varepsilon(\alpha) &= \{\tau^+, *\}\\
\varepsilon(\phi) &= \{\tau^-, state(\tau,\rho)^+, *\}
\end{aligned}
$$

The corresponding rules can be typed as follows. Again, we use subscripts to distinguish the type variables of both agents.

$$
\begin{aligned}
RT(1) &= \{\varepsilon(ret), \varepsilon(\alpha), \{\}\}\\
RT(2a) &= \{\varepsilon(state), \varepsilon(>>=), \{\tau_1 \mapsto \tau_2\}\}\\
RT(2b) &= \{\varepsilon(aux), \varepsilon(\phi), \{\rho_1 \mapsto \rho_2\}\}\\
RT(GRC) &= \{\varepsilon(aux), \varepsilon(ret), \{\rho_1 \mapsto \rho_2\}\}\\
RT(ext) &= \{\varepsilon(pair), \varepsilon(ext), \{\tau_1 \mapsto \tau_2, \rho_1 \mapsto \rho_2\}\}\\
RT(\lambda - abs) &= \{\varepsilon(\lambda), \varepsilon(@), \{\tau_1 \mapsto \tau_2, \rho_1 \mapsto \rho_2\}\}
\end{aligned}
$$

Remarks on the examples We see that both sets of rules are very similar to the textual definition of the *Maybe* and *Writer* monads. In particular, they offer a similar level of abstraction: both rule sets allow any agent as a basic value of the corresponding monad type (Jst, No and log). Thanks to non-uniform port handling of generic rules, any agent with matching port types can be used as second argument of *bind*.

Due to the restriction on interaction rule LHSs, both rulesets feature auxiliary agents and rules. This can be improved by using rules with *nested patterns*: nested patterns are a conservative extension of interaction nets that allows for more complex rule patterns while preserving uniform confluence. For more information, we refer to [12, 11].

6 Discussion

6.1 Related Work

Our approach for a priority constraint in interaction nets was inspired by notions of priority in term rewriting systems [26]. Priorities are a considerable extension to rewriting that can generally violate several properties, for example closure under contexts or substitutions. However, we employ a very restricted form of priorities, only considering priorities of rules on a single redex/active pair. This helps us to maintain the beneficial properties of interaction nets.

Extending interaction nets to a practically usable programming language has been the topic of several publications. One example is [24], where the authors propose a way to represent higher-order recursive functions like *fold* or *unfold*. Clearly, this is a goal very similar to the one of our paper. However, the authors take a different approach than ours. First, they model functions by encoding the (linear) lambda calculus in interaction nets, using specific agents for explicit abstraction and application. While this approach is well-suited to represent higher-order functions, the authors argue that the additional machinery of the encoded lambda calculus complicates the clean representation of terms as nets. Therefore, a second approach is suggested: The authors model higher-order functions through *archetypes*, which can be seen as an abstract structure of interaction rules that can be instantiated with agents representing functions.

We think that the approach of [24] and the ideas presented in the current paper could benefit from each other. First, both archetypes and lambda encodings make use of generic agents. Our results on generic rule constraints could therefore be applied in this setting. Second, archetypes might be a promising basis for a general and abstract framework. They could possibly be used as a unified and extensible interface for monads, similar to the role of *type classes* in Haskell.

Another approach to higher-order functions can be found in [9]: the authors introduce *bigraphical nets*, an extension of interaction nets based on bigraphs. Informally, agents of a bigraphical net can contain (local) subnets, which may interact with the external parts of the net. The authors use bigraphical nets to model non-strict pattern matching (via the ρ-calculus). In addition, the idea of agents containing nets adds a higher-order character to interaction nets. Further research is needed to see whether this approach can be combined with our ideas.

Intersection types for interaction nets [4] had a strong influence on the definition of our rule type system. They have been used to find criteria for termination of interaction nets systems.

Besides in functional programming, monads have been used in several other domains. A recent application can be found in [18], where monads serve for structuring mechanisms in interactive theorem provers.

6.2 Conclusion and Outlook

In this paper, we presented generic rules for interaction nets. These rules are substantially more powerful than ordinary interaction rules and allow for more general pattern matching. This adds a higher-order character to interaction rules. In addition, we defined non-uniform port handling in generic rules which gives us a convenient way to model partially evaluated functions. This extension is conservative: using appropriate constraints, we can ensure that the reduction relation satisfies uniform confluence. Moreover, generic rules only use the core features of interaction nets and do not require any external machinery (such as encodings of the lambda calculus [22] or externally defined programs [8]).

We consider generic rules as a substantial step towards promoting interaction nets to a practically usable programming language. In particular, generic rules can describe monads in a general way that does not rely on specific functions or datatypes. Our rule type system ensures that the matching of generic rules is consistent with the type restrictions of the monadic operators. While our system only uses (parametrized) sorts and type variables on the port level, more complex types are modeled implicitly on the level of agents and rules. Even though this type system fulfills the requirements of the examples in this paper, it is still fairly simple and can of course be refined. For instance, we chose a many-sorted approach here without any form of relation between types. With respect to practical usability, an order-sorted type system is certainly preferable. This will be subject to future work.

Generic rules are an important milestone towards our goal of realizing an abstract framework for handling side effects in interaction nets. While we have shown that individual monads can be defined using these rules, a unified, extensible interface for interaction net monads yet has to be defined.

This will also be addressed in future research. One possible direction is the adaptation of archetypes [24], which we will briefly discussed in the previous subsection. Moreover, we plan to thoroughly investigate other approaches to higher-order functions in interaction nets.

The notion of monads that we used throughout this paper is based on their application to functional programming [28, 17]. Monads were originally defined in category theory. Therefore, it might be interesting to try a more categorial approach to monads in interaction nets. In fact, in a recent publication [3], the author uses notions from category theory to explicitly define interaction rule application and rewriting of nets.

Of course, monads are not the only approach to side effects. In particular, *linear logic* could be employed to handle impure functions in interaction nets: there is a strong relation between both formalisms [19]. The language *Clean* is an example for handling side effects with a flavor of linear logic [1]. Further research will be needed to see if a similar approach can be applied to interaction nets.

Besides these theoretical investigations, we are involved in the development of *inets*, a prototype programming language based on interaction nets [13]. To this date, *inets* supports generic rules with fixed-arity agents and the corresponding constraints. The implementation of variadic agents and non-uniform port handling is currently work in progress.

References

[1] Peter Achten and Rinus Plasmeijer. The ins and outs of clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[2] Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

[3] Marc de Falco. An explicit framework for interaction nets. *CoRR*, abs/1010.1066, 2010.

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

[5] Maribel Fernández and Ian Mackie. From term rewriting systems to generalized interaction nets. *Proceeding of PLILP'96. Programming Languages: Implementations, Logics, and Programs*, 1140, 1996.

[6] Maribel Fernández and Ian Mackie. Interaction nets and term rewriting systems. *Theoretical Computer Science*, 1997.

[7] Maribel Fernández and Ian Mackie. Operational equivalence for interaction nets. *Theoretical Computer Science*, 197, 2003.

[8] Maribel Fernández, Ian Mackie, and Jorge Sousa Pinto. Combining interaction nets with externally defined programs. In *Proc. Joint Conference on Declarative programming (APPIA-GULP-PRODE'01), Évora*, 2001.

[9] Maribel Fernández, Ian Mackie, and François-Régis Sinot. Interaction nets vs. the *rho*-calculus: Introducing bigraphical nets. *Electr. Notes Theor. Comput. Sci.*, 154(3):19–32, 2006.

[10] Abubakar Hassan, Eugen Jiresch, and Shinya Sato. Interaction nets with nested patterns: An implementation. Preliminary proceedings of *10th International Workshop on Rule-Based Programming (RULE 2009), Brasília, Brazil, June 28, 2009.*

[11] Abubakar Hassan, Eugen Jiresch, and Shinya Sato. An implementation of nested pattern matching in interaction nets. *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 21:13–25, 2010. Full version of [10].

[12] Abubakar Hassan and Shinya Sato. Interaction nets with nested pattern matching. *Electr. Notes Theor. Comput. Sci.*, 203(1):79–92, 2008.

[13] The inets project. `http://www.informatics.sussex.ac.uk/research/projects/inets/`.

[14] Eugen Jiresch. Realizing impure functions in interaction nets. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *ICGT*, volume 6372 of *Lecture Notes in Computer Science*, pages 394–396. Springer, 2010.

[15] Eugen Jiresch. Realizing impure functions in interaction nets. `http://www.logic.at/people/jiresch/pub/jiresch_rifins.pdf`, 2011. *ECEASST 38, to appear.*

[16] Eugen Jiresch. Realizing impure functions in interaction nets. full version of [14]. `http://www.logic.at/people/jiresch/pub/jiresch_rifins.pdf`, 2011. submitted to *Electronic Communications of the EASST.*

[17] Simon Peyton Jones and Philip Wadler. Imperative functional programming. *ACM Symposium on Principles of Programming Languages (POPL'02)*, October 1992.

[18] Florent Kircher and Cesar Munoz. The proof monad. *Journal of Logic and Algebraic Programming*, 79:264–277, 2010.

[19] Yves Lafont. Interaction nets. *Proceedings, 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108, 1990.

[20] Yves Lafont. Interaction combinators. *Information and Computation*, 137(1):69–101, 1997.

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

[22] Ian Mackie. Efficient lambda-evaluation with interaction nets. In *RTA*, pages 155–169, 2004.

[23] Ian Mackie. Towards a programming language for interaction nets. *Electronic Notes in Theoretical Computer Science*, 2004.

[24] Ian Mackie, Jorge Sousa Pinto, and Miguel Vilaça. Visual programming with recursion patterns in interaction nets. *ECEASST*, 6, 2007.

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

[26] Masahiko Sakai and Yoshihito Toyama. Semantics and strong sequentiality of priority term rewriting systems. *Theor. Comput. Sci.*, 208(1-2):87–110, 1998.

[27] Philip Wadler. Comprehending monads. *Proc ACM Conference on Lisp and Functional Programming, Nice, ACM.*, June 1990.

[28] Philip Wadler. How to declare an imperative. *ACM Comp. Surveys*, 29(3):240–263, September 1997.