Extending the Interaction Nets Calculus by Generic Rules

Eugen Jiresch*

jiresch@logic.at

Institute for Computer Languages
Vienna University of Technology

We extend the textual calculus for interaction nets by generic rules and propose constraints to preserve uniform confluence. Furthermore, we discuss the implementation of generic rules in the language *inets*, which is based on the lightweight interaction nets calculus.

1 Introduction and Overview

Interaction nets are a model of computation based on graph rewriting. Programs and data are represented as graphs (*nets*), and execution of a program is modeled by manipulating the net based on *rewrite* or *reduction rules*.

The theory behind interaction nets is well developed: they enjoy several useful properties such as uniform confluence and locality of reduction: single computational steps in a net do not interfere with each other, and thus may be performed in parallel. Additionally, interaction nets share computations: reducible expressions cannot be duplicated, which is beneficial for efficiency in computations. Furthermore, the graphical notation of interaction nets automatically provides a visualization of an algorithm. Such a visualization can even show formal properties of programs that might be hard to prove in a textual programming language [16].

Our goal is to promote interaction nets to a practically usable programming language. Unfortunately, the beneficial properties of interaction nets impose strong restrictions on the shape of rules: this makes it hard to express features such as higher-order functions or side effects. In this paper, we improve this deficiency by extending the textual calculus for interaction nets by *generic rules*. In addition, we define constraints on these rules to preserve uniform confluence, which is the basis for parallel evaluation. Despite the merits of the graphical notation, the textual calculus for interaction nets [3] is indispensable: it provides a precise semantics for the mechanics of the graphical rewriting rules. Furthermore, it forms the basis for implementations of interaction nets based languages [7].

We complement our previous work [11], which defined generic rules in the graphical setting of interaction nets. Defining generic rules in the textual calculus gives a precise semantics to the graphical notation which we introduced previously. In addition, we describe the ongoing implementation of generic rules in the interaction nets based language *inets*. We show that the implementation satisfies the constraints for generic rules, and hence preserves uniform confluence. Our contributions can be summarized as follows:

- We extend the interaction nets calculus with generic rules. In particular, we provide a precise definition of *variadic* (arbitrary-arity) rules such as duplication and deletion.
- We describe the implementation of generic rules in the programming language *inets*.

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

S.Alves and I. Mackie (Eds.): Second International Workshop on Linearity 2012 (LINEARITY'12).
EPTCS 101, 2012, pp. 12–24, doi:10.4204/EPTCS.101.2

- We show that the generic rule constraints are satisfied in the calculus and the implementation, ensuring uniform confluence of reduction.

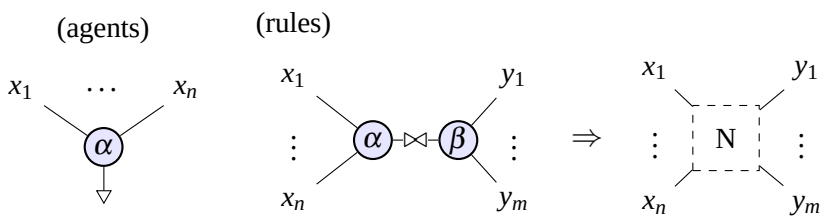
In the following section, we introduce interaction nets and the lightweight calculus. Section 3 defines generic rules for the lightweight calculus. We discuss our ongoing implementation of generic rules in *inets* in Section 4. Finally, we conclude in Section 5.

2 Preliminaries

In this section, we recall the main notions of interaction nets and the lightweight calculus. We discuss the uniform confluence property in both the graphical and the textual formalism. Preserving this property is a key challenge in the introduction of generic rules.

2.1 Interaction Nets

Interaction nets were first introduced in [13]. A *net* is a graph consisting of *agents* (labeled nodes) and *wires* (edges). Each agent has a fixed number of *ports*. Wires connect agents through these ports. We say that an agent is or arity n if it has $n+1$ ports: every agent has exactly one *principal* port (denoted by the arrow), all other ports are called *auxiliary* ports. Intuitively, agent labels denote data or function symbols. 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*, forming an *active pair*. We will refer to a set of rules as *interaction net system* (INS for short). This simple system allows for parallel evaluation of programs: if several rules are applicable at the same time, they can be applied in parallel without interfering with each other. The main prerequisite for this parallelism is the *uniform confluence property* of the reduction relation induced by a set of rules.

Definition 2.1.1 (**Uniform Confluence**)**.** *A relation $\rightarrow$ satisfies the uniform confluence property if the following holds: if $N \rightarrow P$ and $N \rightarrow Q$ where $P \neq Q$, then there exists some R such that $P \rightarrow R \leftarrow Q$.*[1]

Proposition 2.1.2 (Lafont [13])**.** *Let R be an interaction net system. The reduction relation $\Rightarrow$ induced by R satisfies uniform confluence.*

Essentially, three properties of interaction net systems are sufficient for uniform confluence [14]:

1) Linearity: interaction rules cannot erase or duplicate ports.

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

[1] Several publications on interaction nets, including [13], refer to this property as *strong confluence*. We use the term *uniform confluence* (or *WCR_1* in the term rewriting literature [12, 18]) 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.

3) No ambiguity: for each active pair (S,T) of agents there is *at most one* rule that can rewrite (S,T). If S and T are the same agent, then rewriting (S,T) must yield the same net as rewriting (T,S).[2]

Later, we will see how generic rules influence these properties. Essentially, we need to provide constraints on generic rules such that 3) is still satisfied.

2.2 The Lightweight Interaction Calculus

The lightweight calculus [7] provides a precise semantics for interaction nets. It handles application of rules as well as rewiring and connecting of ports and agents. It uses the following ingredients:

Symbols Σ representing agents, denoted by α,β,γ.

Names N representing ports, denoted by $x,y,z,x_1,y_1,z_1,\ldots$. We denote sequences of names by $\overline{x},\overline{y},\overline{z}$.

Terms T being either names or symbols with a number of subterms, corresponding to the agent's arity: $t = x \mid \alpha(t_1,\ldots,t_n)$. s,t,u denote terms, $\overline{s},\overline{t},\overline{u}$ denote sequences of terms.

Equations E denoted by $t = s$ where t,s are terms, representing connections in a net. Note that $t = s$ is equivalent to $s = t$. Δ,Θ denote multisets of equations.

Configurations C representing a net by $\langle \overline{t} \mid \Delta \rangle$. $\overline{t}$ is the interface of the net, i.e., its ports that are not connected to an agent. All names in a configuration occur at most twice. Names that occur twice are called *bound*.

Interaction Rules R denoted by $\alpha(\overline{x}) = \beta(\overline{y}) \longrightarrow \Theta$. α,β is the active pair of the left-hand side (LHS) of the rule and the set of equations Θ represents the right-hand side (RHS).

The *no ambiguity* constraint of Section 2.1 corresponds to the following definition for the lightweight calculus.

Definition 2.2.1 (No Ambiguity). *We say that a set of interaction calculus rules R is* non-ambiguous *if the following holds:*

- *for all pairs of symbols (α,β), there is at most one rule $\alpha(\overline{x}) = \beta(\overline{y}) \longrightarrow \Theta$ or $\beta(\overline{y}) = \alpha(\overline{x}) \longrightarrow \Theta \in R$.*
- *if an agent interacts with itself, i.e., $\alpha(\overline{x}) = \alpha(\overline{y}) \longrightarrow \Theta \in R$, then Θ equals Δ (as multisets, modulo orientation of equations), where Δ is obtained from Θ by swapping all occurrences of $\overline{x}$ and $\overline{y}$.*

Rewriting a net is modeled by applying four *reduction rules* to a configuration with respect to a given set of interaction rules R:

Definition 2.2.2 (Reduction Rules). *The four reduction rules of the lightweight calculus are defined as follows:*

Communication: $\langle \overline{t} \mid x = t, x = u, \Delta \rangle \xrightarrow{com} \langle \overline{t} \mid t = u, \Delta \rangle$

Substitution: $\langle \overline{t} \mid x = t, u = s, \Delta \rangle \xrightarrow{sub} \langle \overline{t} \mid u[t/x] = s, \Delta \rangle$, *where u is not a name and x occurs in u.*

Collect $\langle \overline{t} \mid x = t, \Delta \rangle \xrightarrow{col} \langle \overline{t}[t/x] \mid \Delta \rangle$, *where x occurs in $\overline{t}$.*

Interaction $\langle \overline{t} \mid \alpha(\overline{t_1}) = \beta(\overline{t_2}), \Delta \rangle \xrightarrow{int} \langle \overline{t} \mid \Theta', \Delta \rangle$, *where $\alpha(\overline{x}) = \beta(\overline{y}) \longrightarrow \Theta \in$ R. Θ' denotes Θ where all bound names in Θ receive fresh names and $\overline{x},\overline{y}$ are replaced by $\overline{t_1},\overline{t_2}$.*

[2] See [14] for a detailed explanation of the idea behind the condition for $S = T$.

The reduction rules $\xrightarrow{com}$ and $\xrightarrow{sub}$ replace names by terms: this explicitly resolves connections between agents which are generated by interaction rules. $\xrightarrow{col}$ also replaces names, but only for the interface. Naturally, $\xrightarrow{int}$ models the application of interaction rules: an equation corresponding to a LHS is replaced by the equations of the RHS.

Example 2.2.3. *The rules for addition of symbolic natural numbers are expressed in the lightweight calculus as follows:*

$$+(y,r) = S(x) \longrightarrow +(y,w) = x,\ r = S(w) \tag{1}$$
$$+(y,r) = Z \longrightarrow r = y \tag{2}$$

Proposition 2.2.4 (Uniform Confluence for the Lightweight Calculus). *Let $\longrightarrow$ be the reduction relation induced by the four reduction rules and a set of interaction rules R. If R is non-ambiguous, then $\longrightarrow$ satisfies uniform confluence.*

Proof (sketch). In [3], uniform confluence is shown for the interaction calculus, which is the predecessor of the lightweight calculus. The main difference of the lightweight calculus to the previous one is that the *indirection* rule of the standard interaction calculus is now split into $\xrightarrow{com}$ and $\xrightarrow{sub}$. However, this does not affect the property shown in [3]: all critical pairs (i.e., critical one-step divergences in the reduction of a configuration) can be joined in one step.
It is necessary that R is non-ambiguous in order to prevent non-determinism in the application of the $\xrightarrow{int}$ rule, which could lead to non-joinable divergences. □

3 Generic Rules for the Lightweight Calculus

In this section, we first introduce generic rules in the graphical setting of interaction nets. Afterwards, we extend the lightweight interaction calculus in order to express the semantics of generic rules.

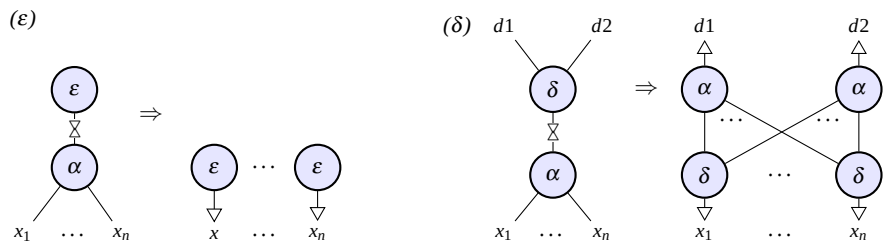
3.1 Generic Rules

Ordinary interaction rules describe the reduction of a pair of two concrete agents (e.g., 0 and + in rule (1) above). *Generic rules* allow one concrete agent to interact with an arbitrary agent. This arbitrary, *generic* agent corresponds to a function variable, adding a higher-order character to interaction nets. Such rules have already been used in several publications (e.g., [15]), usually to model duplication and deletion of agents, albeit without a formal definition of generic rules.
We distinguish two types of generic agents and rules based on the arity of the agent:

fixed generic agents have a specific arity. They correspond to an arbitrary agent of exactly this number of ports.

variadic agents are of arbitrary arity. They correspond to any agent with any number of ports.

Example 3.1.1. *The following rules model deletion and duplication via the agents ε and δ, where α is a* variadic *agent.*



Informally, ε deletes any agent α and propagates itself to α's ports, deleting connected agents in subsequent steps. Similarly, δ duplicates an arbitrary agent and the net connected to it.

The dots at the ports of the variadic agent α indicate that its arity is arbitrary, i.e., any active pair (δ, A) matches this rule (where A may be any agent). While this notation is intuitive, it does not give a precise definition of the semantics of generic rule application. In particular, the RHS of the δ rule has multiple sets of arbitrarily many ports and agents, which may make it more difficult to comprehend. Hence, we provide a definition of generic rule application in the lightweight calculus, clarifying the mechanics that are associated with the graphical dot notation.

3.2 Fixed Generic Rules for the Lightweight Calculus

We first extend the calculus by fixed generic rules. The more complex variadic rules are defined in the following subsection. Essentially, we introduce additional symbols for generic agents. We then modify the $\xrightarrow{int}$ reduction rule to support generic agents.

Generic Names V representing generic agents, denoted by ϕ, ψ, ρ. Generic names may only occur in generic interaction rules.

Generic Rules GR denoted by $\alpha(\overline{x}) = \phi(\overline{y}) \longrightarrow \Theta$. Θ contains no generic names other than ϕ.

The reduction rule for interaction is extended to support matching and application of generic rules.

Definition 3.2.1 (Generic Interaction). $\langle\, \overline{t} \mid \alpha(\overline{t_1}) = \beta(\overline{t_2}), \Delta \rangle \xrightarrow{int} \langle\, \overline{t} \mid \Theta', \Delta \rangle$, *where* $\alpha(\overline{x}) = \beta(\overline{y}) \longrightarrow \Theta \in \mathrm{R}$ *or* $\alpha(\overline{x}) = \phi(\overline{y}) \longrightarrow \Theta \in \mathrm{GR}$ *if* β *and* ϕ *have the same* arity *(number of ports). In the latter case,* Θ' *equals* Θ *where all occurrences of* ϕ *are replaced by* β *(in addition to using fresh names and replacing* $\overline{x}, \overline{y}$*).*

The above definition gives a precise semantics for the application of generic rules with generic agents of fixed arity. Our approach is extended to generic rules with variadic agents in Section 3.4.

Note that the definition of generic interaction only modifies the behaviour of the $\xrightarrow{int}$ rule. The other three reduction rules are not affected by this change: they only operate on configurations, which do not feature generic names.

3.3 Generic Rule Constraints

Unfortunately, generic rules introduce *ambiguity* or *overlaps* to rule application: one equation could possibly be reduced by more than one interaction rule. As mentioned in Proposition 2.2.4, *no ambiguity* is one of the required properties for uniform confluence. Hence, overlaps may destroy the nice properties of interaction nets (including parallel evaluation). Therefore, overlaps caused by generic rules need to be prevented.

In [11], we defined generic rule constraints to preserve uniform confluence in the graphical setting of interaction nets. These constraints can be translated to the lightweight calculus in a straightforward manner. The Default Priority Constraint (DPC) corresponds to a modification of the $\xrightarrow{int}$ reduction rule, just as it restricts the reduction relation in the graphical setting in [11].

Default Priority Constraint (DPC) An equation $\alpha(\overline{t_1}) = \beta(\overline{t_2})$ can only be reduced using a generic rule if no matching ordinary rule exists, i.e., if $\alpha(\overline{x}) = \beta(\overline{y}) \longrightarrow \Theta \notin \mathrm{R}$.

Generic Rule Constraint (GRC) If there is more than one generic rule that can be applied to a given equation $\alpha(\overline{t_1}) = \beta(\overline{t_2})$, there must exist an ordinary rule that can be applied as well.

The DPC restricts the behavior of $\xrightarrow{int}$: ordinary rules always have priority over generic rules. The GRC restricts the set of generic rules *GR*. The combination of these constraints prevents overlaps:

Proposition 3.3.1. *Let R be a set of interaction rules (including generic rules) that satisfies the GRC. If* $\xrightarrow{int}$ *satisfies the DPC, then there is at most one rule that can reduce an arbitrary equation* $\alpha(\overline{s}) = \beta(\overline{u})$.

Proof. We distinguish two possible cases of overlaps:

1. One ordinary and one generic rule can be applied to the same equation (as defined in Definition 3.2.1). Then, the ordinary rule is chosen due to the DPC.
2. There are two generic rules that can be applied to the same equation. Then, by the GRC there must also be an ordinary equation that can be applied. This rule is again prioritized by the DPC.

In both cases, there is only one possible rule that can be applied. As with ordinary interaction rules, the case of two ordinary rules with the same active pair is ruled out. □

With the DPC, a generic rule corresponds to a set of non-ambiguous ordinary rules. The GRC eliminates a few obvious cases of rule overlaps. Analogously to [11](Proposition 3.3.3), we can now show uniform confluence of the lightweight calculus with generic rules.

Proposition 3.3.2 (Uniform Confluence). *Let* $\longrightarrow$ *be the reduction relation induced by the four reduction rules and a set of interaction rules (including generic rules) R that satisfies the GRC. If* $\xrightarrow{int}$ *satisfies the DPC, then* $\longrightarrow$ *satisfies the uniform confluence property.*

Proof (sketch). The main argument is similar to the one used in Proposition 2.2.4: all critical pairs can be joined in one step. Generic interaction rules do not affect the reduction rules $\xrightarrow{col}$, $\xrightarrow{com}$, $\xrightarrow{sub}$. Proposition 3.3.1 shows that the generic rule constraints prevent any ambiguity that might arise from the application of the $\xrightarrow{int}$ rule. □

3.4 Variadic rules for the lightweight calculus

We now extend the lightweight calculus with variadic rules. First, we define additional symbols to denote variadic agents and rules. We exploit the fact that all ports of a variadic agent are handled in the same, *uniform* way when applying a rule.

Clearly, the lightweight calculus needs to capture the feature of arbitrary arity (visualized by the dot notation) in a precise way. Intuitively, arbitrary arity boils down to two mechanisms, as can be seen in the variadic rules for δ/ε in Example 3.1.1:

1. A single agent may have arbitrarily many ports connected to it, like α in the LHS of both rules.

2. A net may be connected to each of the arbitrarily many ports, resulting in arbitrarily many agents. This can be seen in the RHS of the ε-rule, which contains one epsilon for each port.

Note that in case 2), the net connected to each port is the same, i.e., the ports are handled *uniformly*.

These two aspects of variadic rules are captured in the notions of *variadic ranges* and *names*.

Variadic Ranges denoted by $[x],[y],[z],\ldots$, where x,y,z are names. A variadic range corresponds to the set of (arbitrarily many) ports of a variadic agent.

Variadic Names *VN* denoted by $x',y',z',\ldots$. x' denotes an arbitrary single port of the variadic range $[x]$. A variadic name may only appear in the RHS of a rule.

Variadic Rules *VR* are generic rules $\alpha(\overline{y}) = \phi([x]) \longrightarrow \Theta$ where Θ may contain:

- ordinary equations
- equations with variadic ranges
- equations with variadic names

An equation in Θ must not have a variadic range and a variadic name at the same time.

Intuitively, ranges denote the full set of ports of a variadic agent. Variadic names refer to a single port of the variadic agent. All ports of a variadic agent are handled in the same way when applying a rule. Therefore, an equation containing a variadic name specifies how to treat each individual port of the variadic range. As with regular names, variadic ranges and names may appear at most twice in a rule RHS.

Variadic rules capture the two mechanisms mentioned above: the manipulation of a single, arbitrary port and the manipulation of the set of all ports. These two operations are sufficient to provide the expressive power of variadic rules in the graphical setting.

The application of the $\xrightarrow{int}$ reduction rule gets a bit more complicated in the presence of variadic rules: we have to take the arity of the agent corresponding to the varidiac agent into account when replacing an equation.

Definition 3.4.1 (Variadic Interaction). *A variadic interaction step is defined as* $\langle \overline{r} \mid \alpha(\overline{t_\alpha}) = \beta(\overline{u}), \Delta\rangle \xrightarrow{int} \langle \overline{r} \mid \Theta', \Delta\rangle$, *where* $\alpha(\overline{z}) = \phi([x]) \longrightarrow \Theta \in \mathrm{VR}$ *and* Θ' *is instantiated from* Θ *as follows: (let* $arity(\beta) = n$*)*

- *if* $t = \gamma([x]) \in \Theta$, *then* $t = \gamma(\overline{u}) \in \Theta'$.
- *if* $t = \gamma([y]) \in \Theta$ *with* $y \neq x$, *then* $t = \gamma(y_1,\ldots,y_n) \in \Theta'$.
- $t = s \in \Theta$ *such that* t *and* s *contain one or more variadic names* $x',y',\ldots$. *We then add* n *equations* $t_1 = s_1,\ldots,t_n = s_n$ *to* Θ': $t_i = s_i$ $(1 \leq i \leq n)$ *equals* $t = s$ *where all occurrences of a variadic name* x' *are replaced by* u_i *(where* $\overline{u} = u_1,\ldots,u_n$*) if the range* $[x]$ *occurs in the LHS or by the name* x_i *otherwise.*
- *equations without variadic ranges or names are added to* Θ' *without change.*
- *all occurrences of* ϕ *in* Θ *are replaced by* β *and all occurrences of* $\overline{z}$ *by* $\overline{t_\alpha}$.

Informally, a variadic range in Θ is replaced by β's arguments if it occurs in the rule LHS: for example, $[x]$ is replaced by $\overline{u}$ in the above definition. Otherwise, the range is replaced by n fresh names, where $n = arity(\beta)$. An equation with a variadic name is copied to Θ' n times: if a variadic name corresponds to the variadic range in the LHS, it is replaced by one of β's arguments in each copy. Otherwise, it is replaced by one of the fresh names of the corresponding variadic range: for example, $t = y'$ is replaced by $t = y_1,\ldots,t = y_n$, where the names $y_1,\ldots,y_n$ are the result of instantiating the range $[y]$.

Example 3.4.2. *The variadic rule for deletion via the agent ε can be defined as follows: $\varepsilon = \phi([x]) \longrightarrow \varepsilon = x'$. Applying the rule to the equation $\varepsilon = A(u,v,t)$ yields $\{\varepsilon = u, \varepsilon = v, \varepsilon = t\}$: the name x occurs in the LHS (via $[x]$), hence $e = x'$ is instantiated three times for each of A's arguments u,v,t.*

Example 3.4.3. *The variadic rule for duplication via the agent δ can be defined as follows: $\delta(d_1,d_2) = \phi([x]) \longrightarrow d_1 = \phi([y]),\ d_2 = \phi([z]),\ x' = \delta(y',z')$. Applying the rule to the equation $\delta = A(u,v,t)$ yields $\{d_1 = A(y_1,y_2,y_3),\ d_2 = A(z_1,z_2,z_3),\ u = \delta(y_1,z_1),\ v = \delta(y_2,z_2),\ t = \delta(y_3,z_3)\}$: replacing the RHS-only variadic ranges $[y],[z]$ yields the fresh names y_1,y_2,y_3,z_1,z_2,z_3, which are used in the 3 instances of the equation $x' = \delta(y',z')$.*

These examples show that the textual definition of variadic rules is very concise. It clearly expresses the semantics of variadic rules in the graphical setting (see Example 3.1.1). Variadic ranges and names formalize the mechanics expressed by the graphical dot notation.

3.5 Variadic rules with non-uniform port handling

The main characteristic of variadic rules is that every port of a variadic agent is handled in the same way, i.e., connected to the same agent or identical nets. This is strongly connected to the notion of arbitrarily many ports, making them in a sense indistinguishable. For fixed generic agents, we can of course distinguish between their ports and handle them in different ways during rule application.

Both of these aspects of generic rules can be combined in the form of *non-uniform port-handling* [11]. In addition to their arbitrarily many ports, variadic agents may have a fixed, finite number of ports which may be handled specifically, or non-uniformly. Such a variadic agent matches all agents with an arity greater or equal to the number of fixed ports.

This feature translates well to the lightweight calculus. Handling the fixed ports of the variadic agent is expressed with ordinary equations. Definition 3.4.1 in the previous subsection states that only equations with variadic ranges or names are treated in a special way. Ordinary equations are handled the same way as in the ordinary $\xrightarrow{int}$ rule. This means that the fixed ports are independent of the set of arbitrarily many ports. As an example, we recall the rules of the *Maybe* monad from [11].

Example 3.5.1. *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
```

In the lightweight calculus, the Maybe monad is expressed by these rules (ε is defined in Example 3.4):

$$
\begin{aligned}
Ret(r) = \phi([x]) &\longrightarrow \{r = Jst(\phi([x]))\} && (1)\\
Jst(a) = \ >>=(b) &\longrightarrow \{a = b\} && (2)\\
No = \ >>=(b) &\longrightarrow \{Aux = b\} && (3a)\\
Aux = \phi(r,[x]) &\longrightarrow \{\varepsilon = x', No = r\} && (3b)\\
Aux = ret(r) &\longrightarrow \{No = r\} && (GRC)
\end{aligned}
$$

The rules are labeled in correspondence to the lines of Haskell's Maybe monad definition. The (GRC) rule is added to satisfy the generic rule constraint, eliminating ambiguity between rules (1) and (3b). In rule (3b), ϕ has both a variadic range $[x]$ and a single port r which is handled non-uniformly. Just like

f in the Haskell definition, this rule accepts an arbitrary function, including partially applied (curried) functions, e.g., `(+1)`.

Example 3.5.2. *Consider a function* `pick`*, which picks the nth element of a list or returns* `Nothing` *if that element does not exist:*

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

The corresponding interaction rules can be defined as follows (the arguments are swapped for better readability of the rules):

$$
\begin{aligned}
pick(r,n) = Nil &\longrightarrow \{r = No, \varepsilon = n\} && (1)\\
pick(r,n) = Cons(x,xs) &\longrightarrow \{pickH(r,x,xs) = n\} && (2)\\
pickH(r,x,xs) = Z &\longrightarrow \{r = Jst(x), \varepsilon = xs\} && (3)\\
pickH(r,x,xs) = S(n) &\longrightarrow \{pick(r,n) = xs, \varepsilon = x\} && (4)
\end{aligned}
$$

Using the rules for the Maybe *monad from the previous example, we can evaluate the expression* `Nothing >>= (pick 0)`*:*

$$
\begin{aligned}
\langle\, r \mid No = \ >>=(f), f = pick(r,Z)\,\rangle &\xrightarrow{int} \langle\, r \mid Aux = f, f = pick(r,Z)\,\rangle \xrightarrow{com} \langle\, r \mid Aux = pick(r,Z)\,\rangle\\
&\xrightarrow{int} \langle\, r \mid No = r, \varepsilon = Z\,\rangle \xrightarrow{int} \langle\, r \mid No = r\,\rangle \xrightarrow{col} \langle\, No \mid \ \rangle
\end{aligned}
$$

We conclude this section with an example on how to use generic rules to model higher-order functions. We describe the well-known `map` function, which is modeled in a similar way to the monadic operator $>>=$.

Example 3.5.3. *The function* map *is defined as follows:*

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

Using variadic rules with non-uniform port handling, we can define map *in the lightweight calculus. Since the* `f` *is deleted and duplicated in the definition above, we again use* ε *and* δ *agents.*

$$
\begin{aligned}
map(r) = Nil &\longrightarrow \{mapN = r\} && (1)\\
map(r) = Cons(a,as) &\longrightarrow \{mapC(a,as) = r\} && (2)\\
mapN = \phi(r,[x]) &\longrightarrow \{Nil = r, \varepsilon = x'\} && (3)\\
mapC(a,as) = \phi(r,[x]) &\longrightarrow \{Cons(s,t) = r, \phi(s,[y]) = a, \phi(t,[z]) = u,\\
&\qquad map(u) = as, \delta(y',z') = x')\} && (4)
\end{aligned}
$$

4 Implementation

In this section, we discuss the ongoing implementation of generic rules in the prototype language *inets*[9], which is based on the interaction nets calculus. We can show that our implementation satisfies the generic

rule constraints defined in Section 3.3. We will only describe the implementation of fixed generic rules, as the implementation of variadic rules is still work in progress.

inets consists of two components: the *inets* language and compiler, and the runtime system. The *inets* language is based on the interaction calculus, and is compiled to C code, which is executed by the runtime system. The runtime holds data structures for managing interaction rules as well as agents and connections between them. For example, the interaction rules for addition of natural numbers is implemented by the following piece of code (We use the syntax of the *inets* language, where equations are denoted by >< on the LHS, and by ∼ on the RHS):

```
Add(r, y) >< Z   =>  r∼z;
Add(r, y) >< S(x)  =>  r∼S(w), x∼Add(w, y);
```

The implementation of generic rules allows us to define interaction net systems similar to the Maybe monad in Example 3.5.1 (the keyword `ANY` denotes a generic agent):

```
Return(r) >< ANY(x)  => r∼Just(ANY(x))
Bind(r)   >< Just(x) => r∼x;
Bind(r)   >< Nothing => r∼Aux;
Aux       >< ANY(r)  => r∼Nothing;
```

For a complete and detailed description of *inets* and its runtime, we refer to [6]. Here, we will concentrate on the extension of the matching function to support generic rules.

4.1 Generic Rule Constraints

Individual interaction rules (both ordinary and generic) are represented as C functions that take references of the two agents of an active pair as arguments. These functions replace an active pair by the corresponding RHS net and connect it to the rest of the net accordingly. The runtime maintains a table that maps a pair of agent symbols to an interaction rule function. This table also contains entries for fixed generic rules, with a special symbol for generic agents of a specific arity. The following pseudocode describes the matching and reduction function, where ϕ_n denotes the generic agent of arity n:

```
void reduce(agent1, agent2) {
  // is there an ordinary rule for the active pair?
  rulePtr rule = ruleTable[agent1][agent2]
  if (rulePtr == null) {
      // is there a fixed generic rule matching the pair?
      bool success = reduceGeneric(agent1, agent2)
      if (!success)
        error("no matching rule!")
  }
  else {
    rule(agent1, agent2) //apply the ordinary rule
  }
}

bool reduceGeneric(agent1, agent2) {
  //is there a fixed generic rule with matching arity?
  n = arity(agent1)
```

```
  m = arity(agent2)
  rulePtr rule = ruleTable[φn][agent2]
  if (rule == 0) {
    rule = ruleTable[φm][agent1]
    if (rule == 0)
      return false  // no matching generic rule
  }
  // apply the generic rule
  rule(agent1, agent2)
  return true
}
```

The Generic Rule Constraint (GRC) is a property of the set of interaction rules, and can thus be verified at compile time. We check each generic rule for overlaps with already compiled generic rules, as shown by the following pseudocode:

```
void checkGRC(Rule r) {
  if (r is a generic rule) {
    let A be the ordinary agent of r′s LHS
    let φm be the generic agent of r′s LHS
    n = arity(A)
    if (a generic rule with LHS B >< φn exists
     and arity(B) = m ) {
      // we have two overlapping generic rules
      if (no ordinary rule with LHS A >< B exists)
        error("generic rule overlap!")
    }
    add r to the existing rules
  }
}
```

Clearly, the implementation needs to satisfy the generic rule constraints of Section 3.3. Otherwise, multiple generic rules may overlap, resulting in non-determinism in the evaluation of the program. It is straightforward to see that the pseudocode above satisfies the generic rule constraints:

Proposition 4.1.1. *The implementation of generic interaction rules in* inets *satisfies the DPC and GRC.*

Proof. Consider the function `reduce`. The application of a generic rule via `reduceGeneric` is only attempted if no ordinary interaction rule exists. Hence, the DPC is satisfied. For the GRC, consider `checkGRC`: if the generic rule currently being checked overlaps with a previous generic rule and no matching ordinary rule exists (i.e., the GRC is violated), an error is reported. □

Implementation of Variadic Rules The current version of *inets* also supports variadic rules. During compilation, a variadic rule is translated into a set of fixed generic rules, one for each possible arity. While variadic rules are theoretically defined for agents with arbitrarily large arities, we can easily identify the maximum arity n of all agents in the current program. Hence, we only add n fixed versions of the variadic rule. After this step, the set of rules only contains fixed generic rules, which are handled as described above.

5 Discussion

5.1 Related Work

The textual calculus for interaction nets was initially defined in [3]. We based our extensions on the improved *lightweight* calculus, which was introduced in [7]. A different approach to higher-order computation in the interaction calculus can be found in [4].

Besides *inets*, several implementations of interaction nets evaluators exist. Examples are *amineLight* [7] or INblobs [1]. To the best of our knowledge, none of these systems support generic interaction rules. Another recent tool is PORGY [2], which can be used to analyse and evaluate interaction net systems with a focus on evaluation strategies.

The extension of interaction nets to a practically usable programming language has been the topic of several publications. For example, in [17] the authors propose a way to represent higher-order recursive functions like *fold* or *unfold*. *Nested patterns*, an extension to allow more complex interaction rules, have been dealt with in [5, 8]. They combine well with generic rules.

5.2 Conclusion and Further Work

In this paper, we extended the interaction nets calculus by generic rules. Our previous work on generic rules [11] did not consider this textual calculus. Instead, we defined generic rules and their constraints (including a basic type system) in the graphical setting. The extension of the lightweight calculus provides an alternative precise semantics to the graphical notation of generic interaction rules. This is particularly important for variadic rules, which use a dot notation to express an arbitrary number of ports. Our approach using variadic names and ranges is concise and precisely formulates the mechanics of the dot notation of the graphical rewrite rules.

In addition, we discussed the ongoing implementation of generic rules in *inets*. This implementation satisfies the generic rule constraints DPC and GRC and hence preserves uniform confluence.

Generic rules allow us to conveniently express higher-order functions. An example can be found in [10], where an interaction nets encoding of *map* is given. Generic agents assume a role similar to function variables in functional programming. Hence the definition of higher-order functions via interaction rules closely mimics functional programs without the need for explicit lambda and function application agents. This brings interaction nets closer to a practically usable programming language.

A major motivation for formally dealing with generic rules comes from our own previous work [10, 11], where we used generic rules to model side effects in interaction nets via monads. As part of future work, we plan to define an abstract, unified interface for monads, similar to type classes in Haskell. The *agent archetype* approach of [17] may be a possible direction to achieve this. In addition, we will continue to contribute to *inets*. Moreover, we are currently investigating an implementation of interaction nets on parallel hardware (GPUs).

References

[1] Jose Bacelar Almeida, Jorge Sousa Pinto & Miguel Vilaca (2008): *A tool for programming with interaction nets*. *Electronic Notes in Theoretical Computer Science* 219, pp. 83–96, doi:10.4204/EPTCS.48.7.

[2] Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet & Bruno Pinaud (2011): *PORGY: Strategy-Driven Interactive Transformation of Graphs*. In: *TERMGRAPH*, pp. 54–68, doi:10.4204/EPTCS.48.7.

[3] Maribel Fernández & Ian Mackie (1999): *A Calculus for Interaction Nets*. In Gopalan Nadathur, editor: *PPDP, LNCS* 1702, Springer, pp. 170–187, doi:10.1007/10704567_10.

[4] Maribel Fernández, Ian Mackie & Jorge Sousa Pinto (2007): *A Higher-Order Calculus for Graph Transformation*. *Electr. Notes Theor. Comput. Sci.* 72(1), pp. 45–58, doi:10.1017/S0960129598002679.

[5] Abubakar Hassan, Eugen Jiresch & Shinya Sato (2010): *An Implementation of Nested Pattern Matching in Interaction Nets*. *Electronic Proceedings in Theoretical Computer Science (EPTCS)* 21, pp. 13–25, doi:10.4204/EPTCS.21.2.

[6] Abubakar Hassan, Ian Mackie & Shinya Sato (2009): *Compilation of Interaction Nets*. *Electr. Notes Theor. Comput. Sci.* 253(4), pp. 73–90. Available at `http://dx.doi.org/10.1016/j.entcs.2009.10.018`.

[7] Abubakar Hassan, Ian Mackie & Shinya Sato (2010): *A lightweight abstract machine for interaction nets*. *ECEASST* 29.

[8] Abubakar Hassan & Shinya Sato (2008): *Interaction Nets With Nested Pattern Matching*. *Electr. Notes Theor. Comput. Sci.* 203(1), pp. 79–92. Available at `http://dx.doi.org/10.1016/j.entcs.2008.03.035`.

[9] *The Inets Project site [Online, accessed 20-December-2011]*. `http://gna.org/projects/inets`.

[10] Eugen Jiresch (2011): *Realizing Impure Functions in Interaction Nets*. *ECEASST 38*.

[11] Eugen Jiresch & Bernhard Gramlich (2011): *Realizing Monads in Interaction Nets via Generic Typed Rules* . `http://www.logic.at/staff/gramlich/papers/techrep-e1852-2011-01.pdf`. Technical Report.

[12] J.-W. Klop (1992): *Term Rewriting Systems*, chapter 1, pp. 2–117. 2, Oxford University Press, doi:10.1.1.35.425.

[13] Yves Lafont (1990): *Interaction Nets*. *Proceedings, 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pp. 95–108, doi:10.1145/96709.96718.

[14] Yves Lafont (1997): *Interaction Combinators*. *Information and Computation* 137(1), pp. 69–101, doi:10.1006/inco.1997.2643.

[15] Ian Mackie (1998): *YALE: Yet Another Lambda Evaluator Based on Interaction Nets*. *International Conference on Functional Programming (ICFP'98)*, pp. 117–128, doi:10.1145/291251.289434.

[16] Ian Mackie (2010): *A Visual Model of Computation*. In Jan Kratochvíl, Angsheng Li, Jirí Fiala & Petr Kolman, editors: *TAMC, Lecture Notes in Computer Science* 6108, Springer, pp. 350–360, doi:10.1007/978-3-642-13562-0_32.

[17] Ian Mackie, Jorge Sousa Pinto & Miguel Vilaça (2007): *Visual Programming with Recursion Patterns in Interaction Nets*. *ECEASST* 6. Available at `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/66`.

[18] Joachim Niehren (2000): *Uniform confluence in concurrent computation*. *J. Funct. Program.* 10(5), pp. 453–499, doi:10.1.1.140.7218. Available at `http://journals.cambridge.org/action/displayAbstract?aid=59755`.