FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Extending Interaction Nets towards the Real World

PhD THESIS

submitted in partial fulfillment of the requirements of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

Eugen Jiresch
Registration Number 0204097

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Inf. Dr.rer.nat. Bernhard Gramlich

External reviewers:
Ian Mackie. LIX, École Polytechnique, France.
Jorge Sousa Pinto. Departamento de Informática, Universidade do Minho, Portugal.

Wien, 31.08.2012

(Signature of Author) (Signature of Advisor)

Declaration of Authorship

Eugen Jiresch
Habichergasse 7/30, 1160 Vienna

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

(Place, Date)	(Signature of Author)

Acknowledgements

The biggest acknowledgement undoubtedly goes to my supervisor Bernhard Gramlich, whose invaluable support made this thesis possible. I am deeply thankful for his dedication in following, evaluating and discussing my work for years, up to sacrificing many late evenings and weekends for my cause.

A big thanks goes to Ian Mackie for introducing me to the topic of interaction nets, and helping me to connect to its scientific community. Without his support, my phd project would certainly not have succeeded. Likewise, my co-authors Bernhard Gramlich, Abubakar Hassan and Shinya Sato contributed a great deal to my work. The collaboration with them helped me gain new insights and develop further ideas.

For their financial support, I would like to thank the Austrian Academy of Sciences (ÖAW) and the Vienna PhD School of Informatics (in particular the school's director, Hannes Werthner).

For small suggestions that had considerable impact, I would like to thank Andreas Dielacher, Christoph Roschger, Karl Gmeiner and Michael Wimmer.

Furthermore, I am very grateful to my parents for their continued support in every sense, in particular their encouragement to find and pursue my own interests.

Finally, I want to thank my wife Maria for making my life a so much better experience and for enduring my bad mood when progress on my PhD was slow.

I dedicate this thesis to anyone with the perseverance to achieve their goals.

Abstract

Interaction nets are a formal model of computation. They use graphs and graph replacement (or rewriting) rules to describe computation. Compared to prominent models such as *Turing machines* or the *λ-calculus*, interaction nets offer two main differences "out of the box": *visualization* and *parallel evaluation*. Rewriting rules and input graphs are given in a graphical notation, visualizing an algorithm. Due to strong restrictions on the shape of these rewriting rules, any interaction nets program can be evaluated in parallel.

The above properties indicate that interaction nets could be a useful foundation for a programming language. To ensure correctness of parallel programs, a formal model that features parallel evaluation as a "first-class citizen" could be indispensable. Unfortunately, interaction nets are a very basic and restricted model of computation: similar to the λ-calculus, it is not feasible to specify practical, real-world applications with the basic model. Interaction nets lack many features of a practical programming language needed to conveniently express complex functions.

The goal of this thesis is to bring interaction nets closer to real-world usability. In order to tackle *computational side effects* (I/O, state manipulation,...), we define *monads* in interaction nets, which are based on *generic interaction rules*. In combination with *nested patterns*, these rules allow us to conveniently define higher-order functions directly in interaction nets. We show that these extensions are conservative: under reasonable constraints, they preserve the beneficial properties of the base model. In addition to these theoretical results, we discuss the implementation of these features and present an approach to realize *parallel evaluation* of interaction nets on graphics processing units (GPUs).

Abstract

Interaction nets sind ein formales Berechnungsmodell: Sie verwenden Graphen und Regeln zur Graphersetzung ("rewriting"), um Berechnung zu beschreiben. Im Vergleich zu etablierten Modellen wie Turingmaschinen oder dem λ-Kalkül bieten interaction nets zwei besondere Eigenschaften: Visualisierung und parallele Evaluierung. Rewriting-Regeln werden in einer grafischen Notation definiert, wodurch Algorithmen visualisiert werden. Durch starke Beschränkungen dieser Regeln kann jedes interaction nets Programm parallel evaluiert werden.

Diese Eigenschaften machen interaction nets zu einer vielversprechenden Basis für eine zukünftige Programmiersprache. Um Korrektheit von parallelen Programmen zu zeigen, wäre ein formales Modell mit paralleler Evaluierung als "first-class citizen" von großem Vorteil. Unglücklickerweise sind interaction nets ein sehr einfaches und eingeschränktes Berechnungsmodell: Ähnlich dem λ-Kalkül ist es nicht praktikabel, ein Programm für reale Anwendungen zu spezifizieren. Interaction nets mangelt es an vielen Features von Programmiersprachen, die zum Erstellen komplexer Funktionen notwendig sind.

Das Ziel dieser Dissertation ist es, interaction nets näher an die Verwendbarkeit als praktikable Programmiersprache zu bringen. Um *Seiteneffekte* (I/O, State manipulation, exception handling,. . .) zu bewältigen, definieren wir *Monaden* in interaction nets, die auf *generic interaction rules* basieren. In Kombination mit *nested patterns* erlauben diese die komfortable Definition von Funktionen höherer Ordnung direkt in interaction nets. Diese Erweiterungen sind konservativ: Mit annehmbaren Einschränkungen erhalten diese die günstigen Eigenschaften des Basismodells. Zusätzlich zu diesen theoretischen Resultaten behandeln wir die Implementierung dieser Features und präsentieren einen Ansatz, um parallele Evaluierung von interaction nets mittels Grafikprozessoren (GPUs) zu realisieren.

Contents

CHAPTER 1

Introduction and Motivation

1.1 Motivation and Background

You are a very fine person, Mr. Baggins, and I am very fond of you; but you are only quite a little fellow in a wide world after all!

J.R.R. Tolkien, The Hobbit

In Tolkien's book *The Hobbit*, the titular character Bilbo Baggins embarks on an adventure which takes him through many regions of the world he lives in. The result of his interaction with the world is two-fold: not only does he change the world with his actions (such as defeating monsters and stealing treasure), but the world also changes him. His journey transforms Mr. Baggins from an average citizen into an adventurer and master thief. The quote above summarizes a message of the book: The outside world is vast and unpredictable, and interacting with it may change you in unexpected ways.

What holds for Mr. Baggins, also holds for computer programs. Modern software frequently interacts with the real world. Programs receive input from a multitude of sensors and devices, output results to displays or communicate over networks. These interactions have a strong, hardly predictable impact on the results of programs, making it difficult to reason about their correctness or guarantee that they are free from errors. Indeed, *formal verification of software* has become increasingly important in the last years. The possibility to show that a program will never crash or perform any unwanted side effects is not restricted to software systems with a direct impact on life or death (e.g., in medicine). Nowadays, more mundane programs such as drivers for end consumer computer hardware are subject to formal verification. Hence, programming languages based on rigorous formal models are indispensable. A prominent example is Haskell, a functional programming language based on the λ-calculus.

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

While the reasons above demonstrate great potential for interaction nets, the existing prototype languages based on interaction nets lack important features for practical use: these include means of interaction with the "real world": input/output functionality, state manipulation or exception handling. Such features of a program changing the outside world (and vice versa) are referred to as *computational side effects* or *impure functions*. In contrast, *pure* (mathematical) functions do not incorporate side effects. Interaction nets can be considered a pure language: the reduction of a net is not influenced by anything but its initial state, and no side effects are performed. Since impure functions are considered a vital part of today's programs, pure languages need to incorporate them. However, computational side effects may generally destroy the beneficial properties of interaction nets. Hence, we require an appropriate interface to deal with impure functions.

Moreover, the existing implementations of interaction nets hardly offer any means for parallel evaluation. This is a striking shortcoming: one of the most compelling features of interaction nets is that parallel evaluation of programs comes "for free". Any program expressed in interaction nets and rules can be executed in parallel. This makes an implementation of interaction nets on a multicore architecture attractive.

However, this potential for parallelism cannot be trivially leveraged in an implementation. The potential for parallelism in an interaction net is highly dynamic, and depends on the particular program and even runtime values. At any point during a computation, the number of expressions that can be evaluated in parallel can vary between dozens and hundreds of thousands. There is currently no implementation of interaction nets that leverages their full parallelism potential.

1.2 Contributions / Aim of the Work

In this thesis, we aim to bring interaction nets closer to "real world" applications: we add new features to the computational model that allow more powerful and convenient rule definitions. These enable impure functions and computational side effects without sacrificing its formal properties. In addition, we provide an implementation of this extension and introduce a prototype for parallel evaluation of interaction nets.

To combine pure functions and the outside world, we extend interaction nets by introducing a framework to handle side effects. This framework is based on *monads*, a model to structure computations that has been used in Haskell with great success. However, there are two main obstacles when adapting monads to interaction nets: first, interaction nets need to support abstract datatypes in order to implement monads. Existing type systems ([8, 30]) for interaction nets are currently lacking this feature. Second, monadic functions have a distinct higher-order character. Even though interaction nets seem well-suited to incorporate higher-order functions (both data and computation are represented as nets and treated equally), it is not possible to conveniently express well-known higher-order functions such as *map* or *fold/reduce*. We address this shortcoming by introducing *generic rules*. These extend ordinary interaction rules and add a form of higher-order functions to the base formalism, thus enabling the definition of monads. Furthermore, we combine generic rules with *nested patterns*, another extension that allows for more complex and practically usable rule definitions.

Improving the formal model of interaction nets is only one side of the coin that buys you practicality. Existing prototype languages need to be improved with regard to language features and performance. We address the former by implementing our theoretical advances in the interaction nets based language *inets*. In order to leverage the potential for parallel evaluation of interaction nets, we provide the prototype implementation *ingpu*. This prototype performs almost all computation on the graphics processing unit (GPU) as opposed to the CPU: in recent years, a trend towards using GPUs for general purpose computations has emerged. Due to the increasing programmability of GPUs and general purpose APIs (CUDA, OpenCL), the parallel processing power of graphics cards may be used for many kinds of problems. While the GPU model of parallelism seems to be well-suited for interaction nets, several factors make an implementation a non-trivial task. We argue that these factors can be overcome, thus enabling an efficient evaluation of interaction nets, provided that a computation can be sufficiently parallelized.

Our main contributions can be summarized as:

1. We introduce generic rules for interaction nets. These rules allow us to define higher-order functions directly in the formal model. This extension is conservative. We impose constraints on generic rules that restrict their usage in some cases, but retain most of their power and - most importantly - preserve the beneficial properties of interaction nets. We define generic rules in the graphical formalism of interaction nets as well as in the textual interaction calculus. The latter provides important foundations for both prototype implementations *inets* and *ingpu*.

2. We introduce the concept of *monads* to interaction nets. We show how monads can be used to model computational side effects directly in interaction nets while preserving the purity of the system. We realize monads by using generic rules and a type system powerful enough to express monadic operators.

3. Using our formal results on the interaction calculus, we describe the implementation of generic rules in *inets*. All theoretical aspects of our rule extension are realized in practice.

4. We show how interaction rules with *nested patterns* can be implemented to allow for more complex rule definitions. We discuss the formal algorithm to translate nested rules to sets of ordinary ones and its implementation in *inets*. In addition, we show that generic rules and nested patterns can be combined elegantly.

5. We describe the ongoing implementation of the maximally parallel interaction nets evaluator *ingpu*. This evaluator runs almost entirely on the GPU and uses recent insights on the efficient implementation of dynamic parallel computation. We describe several

approaches to the implementation, discuss their pros and cons, and present benchmark results that compare *ingpu* to existing interaction nets systems.

We see our work as a considerable step towards turning interaction nets into a practically usable programming paradigm. Future language extensions can be built upon our results, further extending the power and usability of interaction nets.

1.3 Structure of the Thesis

The remainder of this document consists of six chapters. We begin with an introduction to the *state of the art* in Chapter 2. First, we familiarize the reader with the basic definition of (abstract) rewriting systems. We then use these notions to introduce the main formalism of the thesis, interaction nets (INs for short). We discuss their basic properties as well as advantages and disadvantages of the model. The second part of Chapter 2 deals with computational side effects in pure functional programming languages. We describe the notion of a monad in functional programming, which will later be adapted to INs.

Chapter 3 contains the *foundations and theoretical results* of our work. We begin with the introduction of generic interaction rules, which allow for a form of higher-order functions in interaction nets. The power of these rules can potentially destroy the beneficial properties of interaction nets. Therefore, we impose constraints on generic rules to preserve these properties, most importantly uniform confluence. Our discussion of generic rules is split into two parts: we first define generic rules in the graphical setting of interaction nets. Second, we add them to the textual interaction calculus, which is the basis for the implementations of INs. We show the correctness of the extension and preservation of uniform confluence separately for both formalisms. We continue with the definition of a type system for interaction nets that is as simple as possible, yet complex enough to support monadic datatypes and operators.

We conclude Chapter 3 by defining interaction rules with nested patterns and discuss their relation to ordinary rules. We show that both interaction rule extensions - generic rules and nested patterns - can be combined to allow for even more powerful and convenient rules.

In Chapter 4, we show how the results of the previous chapter can be *applied to handle side effects and impure functions* in interaction nets. We adapt the notion of monads to the interaction net setting and give several examples of concrete IN monads, each of which handles a particular type of side effect. We show that each monad is correct in the sense of being a proper monad and that the set of rules it consists of satisfies the constraints imposed by generic rules.

We turn towards the *implementation* of interaction nets based programming languages in Chapter 5. The first part discusses the implementation of our extensions in the language *inets*: we add both nested patterns and generic rules to the language. In addition, we show that the constraints and formal properties of interaction nets as well as of the extensions in general still hold in the implementation. In the second half of the chapter, we discuss the development of *ingpu*, an experimental GPU-based interaction nets evaluator. The tool fully realizes the theoretical parallelism potential of interaction nets by leveraging the computing power of modern many-core graphics cards. We discuss an initial approach to the implementation and subsequent design optimizations which allow *ingpu* to strongly outperform existing interaction nets languages on highly parallel problems.

Chapter 6 presents a conclusion and critical reflection on our results. Both theory and implementation of our extensions have shortcomings and room for improvement, which can be addressed in future work. Furthermore, we discuss related work and its pros and cons as compared to our approach.

This thesis is based on the following publications:

- 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.

- Eugen Jiresch. Realizing Impure Functions in Interaction Nets. *Electronic Communications of the European Association of Software Science and Technology (ECEASST)*, Volume 38. 2011.

- Eugen Jiresch. Extending the IN Calculus with Generic Rules. Sandra Alves and Ian Mackie (eds.): *Proceedings of the Second International Workshop on Linearity (LINEARITY 2012). Electronic Proceedings in Theoretical Computer Science (EPTCS)*, to appear. ETAPS 2012, Tallinn, Estonia.

- Eugen Jiresch and Bernhard Gramlich. Realizing Monads in Interaction Nets via Generic Typed Rules. Manindra Agrawal, S. Barry Cooper, Angsheng Li (eds.): *Proceedings of the 9th annual conference on Theory and Applications of Models of Computation (TAMC 2012). Lecture Notes in Computer Science (LNCS)*, 7287:509-524, 2012, Beijing, China.

- Eugen Jiresch. Towards a GPU-based Implementation of Interaction Nets. *Proceedings of the 8th International Workshop on Developments in Computational Models (DCM 2012).* 2012, Cambridge, UK. conditionally accepted.

CHAPTER 2

State of the Art

This chapter details the foundations of our work. We begin with an introduction to interaction nets (and rewriting systems in general). We discuss both the graphical formalism of interaction nets as well as the textual interaction calculus. Afterwards, we familiarize the reader with the notion of computational side effects and show how they can be handled in pure (functional) languages. Finally, we conclude the chapter with a discussion of the relation between interaction nets and linear logic.

2.1 Rewriting Systems

In this section, we introduce rewriting systems and some core notions that are the basis for interaction nets. We also briefly discuss term rewriting systems, which are one of the most prominent rewriting formalisms.

2.1.1 Abstract Reduction Systems

We define an **abstract reduction system** (**ARS**) as a pair $(A, \rightarrow)$: The reduction $\rightarrow$ is a binary relation on the set A. An ARS can model a step by step activity like a transformation of some object (e.g., a term or a graph) or some other computation. For $(a, b) \in\rightarrow$, we simply write $a \rightarrow b$. The following definitions are within in the context of some arbitrary ARS $(A, \rightarrow)$.

We now introduce some important forms of reduction which are built by **composition**: For two relations $R \subseteq A \times B$ and $S \subseteq B \times C$, their composition is defined by

$$R \circ S := \{(x, z) \in A \times C | \exists y \in B (x, y) \in R \wedge (y, z) \in S\}$$

Definition 2.1.1.1. *We define the following symbols by composing a reduction with itself:*

Based on these symbols, we introduce some additional notations:

$\overset{0}{\rightarrow}$	$:=$	$\{(x,x) \mid x \in A\}$	**identity**
$\overset{i+1}{\rightarrow}$	$:=$	$\overset{i}{\rightarrow} \circ \rightarrow$	**(i+1)-fold composition,** $i \geq 0$
$\overset{+}{\rightarrow}$	$:=$	$\bigcup_{i>0} \overset{i}{\rightarrow}$	**transitive closure**
$\overset{*}{\rightarrow}$	$:=$	$\overset{+}{\rightarrow} \cup \overset{0}{\rightarrow}$	**reflexive transitive closure**
$\overset{=}{\rightarrow}$	$:=$	$\rightarrow \cup \overset{0}{\rightarrow}$	**reflexive closure**
$\overset{-1}{\rightarrow}$	$:=$	$\{(y,x) \mid x \rightarrow y\}$	**inverse**
$\leftarrow$	$:=$	$\overset{-1}{\rightarrow}$	**inverse**
$\leftrightarrow$	$:=$	$\rightarrow \cup \leftarrow$	**symmetric closure**
$\overset{+}{\leftrightarrow}$	$:=$	$(\leftrightarrow)^+$	**transitive symmetric closure**
$\overset{*}{\leftrightarrow}$	$:=$	$(\leftrightarrow)^*$	**reflexive transitive symmetric closure**

1. x is **reducible** if there is a y such that $x \rightarrow y$.
2. x is **in normal form** (**irreducible**) if it is not reducible.
3. y is **a normal form of** x if $x \overset{*}{\rightarrow} y$ and y is in normal form. If x has a uniquely determined normal form, the latter is denoted by $x\downarrow$.
4. y is a **direct successor** of x if $x \rightarrow y$.
5. y is a **successor** of x if $x \overset{+}{\rightarrow} y$.
6. x and y are **joinable** if there is a z such that $x \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y$, which is also written as $x \downarrow y$.

We can now define two core properties of ARSs:

Definition 2.1.1.2. ***(confluence and termination)*** *A reduction relation $\rightarrow$ is called*
confluent *if $y_1 \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y_2 \Rightarrow y_1 \downarrow y_2$ (see Fig. 2.1).*
Church-Rosser *if $x \overset{*}{\leftrightarrow} y \Rightarrow x \downarrow y$ (see Fig. 2.1).*
terminating *if there is no infinite reduction sequence $a_0 \rightarrow a_1 \rightarrow \ldots$*
normalizing *if every element has a normal form.*
convergent *if it is both confluent and terminating.*

$$\begin{tikzcd}
x \arrow[rr, leftrightarrow, "*"] \arrow[dr, dashed, "*"'] & & y \arrow[dl, dashed, "*"] \\
& z &
\end{tikzcd}
\qquad
\begin{tikzcd}
x \arrow[r, "*"] \arrow[d, "*"'] & y_1 \arrow[d, dashed, "*"] \\
y_2 \arrow[r, dashed, "*"'] & z
\end{tikzcd}$$

Figure 2.1: The Church-Rosser property and confluence

The intuition behind confluence is the following: If some object x can be reduced to two different objects, these objects have a common successor, i.e., both can be reduced to the same object.

It turns out that the Church-Rosser property and confluence are equivalent. The fact that any Church-Rosser relation is confluent is almost immediate. The reverse implication can be shown by observing that any equivalence $x \overset{*}{\leftrightarrow} y$ can be seen as a series of peaks between x and y . We can now use confluence to close each of the peaks from the top to the bottom and thus join x and y.

Uniform confluence

Confluence considers divergences of arbitrary length $y_1 \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y_2$. This means that it takes an *arbitrary number of steps* from x to reach y_1 and y_2. Moreover, it may take an arbitrary number of steps from both y_1 and y_2 to reach a common successor z. Intuitively, reducing an object in a confluent ARS means that if there are multiple reduction paths, then we can *eventually* reach a common successor from any path. The emphasis is on “eventually”, as these reduction paths may be arbitrarily long. *Uniform confluence* restricts this arbitrary length of reduction paths to exactly 1:

Definition 2.1.1.3 (Uniform confluence). *A reduction relation $\rightarrow$ is* uniformly confluent *if the following holds: if $y_1 \leftarrow x \rightarrow y_2$ where $y_1 \neq y_2$, then there exists a z such that $y_1 \rightarrow z \leftarrow y_2$.*

In other words, if x has two *distinct one-step* successors in a uniformly confluent ARS, then a common successor to these two objects can be reached by exactly one step from each object. Uniform confluence not only implies confluence, but is strictly stronger. It guarantees a common successor after exactly one step on each side, as opposed to arbitrarily many.

In the following section, we will show that this property is crucial to interaction nets, as it ensures that single computation steps are independent of each other.

2.1.2 Term Rewriting Systems

A *term rewriting system* (TRS) [28] is an abstract reduction system where the set A is built from terms over a given signature Σ containing *function symbols* of a fixed arity and a set of *variables* X. We then denote the set of terms built from these objects by $T(\Sigma, X)$.

Terms can be seen as trees where each node is labeled by a function symbol or a variable, and each non-variable node has a number of child nodes corresponding to its arity. We can then specify positions of subterms by strings of integers: the subterm of t at some position p is denoted by $t|_p$. For example, let $t = f(x, g(f(b, c)))$. Then, $t|_{21}$ refers to $f(b, c)$ and $t|_{212}$ refers to c. For replacing a term $t|_p$ with u in t at position p, we write $t[u]_p$. Furthermore, we define *substitutions* as mappings from variables to terms.

Definition 2.1.2.1. *Let Σ be a signature and X a set of variables. A* Term Rewriting System (TRS) *is a set of rewrite rules $R = l_i \rightarrow r_i$, where $l_i, r_i \in T(\Sigma, X), l_i \notin X$ and all variables that occur in r_i also occur in l_i.*

A term t is rewritten *to a term u by a rule $l \to r$ at position p with substitution σ if $t|_p = l\sigma$ and $u = t[r\sigma]_p$.*

Example 2.1.2.2. *The addition of symbolic natural numbers (modeled by 0 and a successor symbol S) can be defined by a TRS based on the signature $\Sigma = \{0^0, S^1, +^2\}$ and the following rewrite rules:*

$$+(0, y) \to y$$
$$+(S(x), y) \to S(+(x, y))$$

As an example reduction, consider $1 + 1 \to^* 2$:

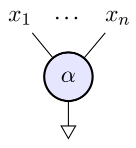
$$+(S(0), S(0)) \to S(+(0, S(0))) \to S(S(0))$$

2.2 Interaction Nets

Interaction nets are a formal model of computation introduced by Yves Lafont in 1990 [30]. They are a form of graph rewriting, which is essentially a specific form of ARS: Programs and data are specified as graphs, and execution or evaluation is modeled by rewriting parts of the graph according to reduction rules. If no more rules are applicable to the graph, a normal form is reached, which corresponds to the result of the computation.

Interaction nets consist of the following components:

- A set of *agents*, labeled by a set Σ of symbols. Each agent is associated with an arity n, describing the number of *ports* connected to the agent. An agent α with arity n has $n + 1$ ports, one of which is denoted the *principal port* whereas all others are *auxiliary ports*.

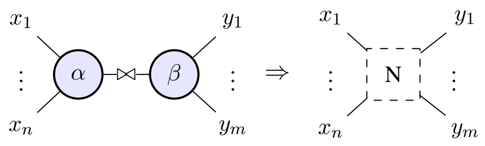
  In this example, the agent labeled α has the principal port x_0 (indicated by the port's arrow) and the auxiliary ports $x_0 \ldots x_n$.

  We can also represent this agent textually as $\alpha[x_1, \ldots, x_n]$ or simply α.

- A *net* built on Σ is an undirected graph with agents at the vertices. The edges of the net connect agents at the ports such that there is only one edge at every port. A port which is not connected is called a *free port*. A set of free ports is called an *interface*.

- An *active pair* or *redex* consists of two agents $(\alpha, \beta) \in \Sigma \times \Sigma$ that are connected via their principal port. Active pairs are the parts of a net that can be rewritten, similar to a *redex* (reducible expression) in other models of computation. The two nodes of an active pair are also called *active agents*.

- *Interaction rules* replace an active pair by some net N. Interaction rules preserve the *interface* of the net: no auxiliary ports are added or removed.

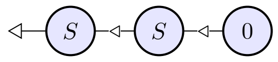


We will represent interaction rules textually as $A \bowtie B \Rightarrow N$, where $A \bowtie B$ represents the active pair on the left-hand side (LHS), and N the net on the right-hand side (RHS). If the RHS is not of importance for an argument, we may just write $A \bowtie B$. We write $A \sim B$ to 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 2.2.1 (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 $A, B \in \Sigma$. There is at most one rule for every such active pair.*
We will often write R instead of $\mathcal{R}$, thus leaving the signature implicit.

Definition 2.2.2 (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 $\Rightarrow_R$.

We now give a simple example modelling the addition of natural numbers. Numbers are represented by the agents 0 (zero) and S (successor). For example, we can express 2 as *the successor of the successor of zero*, or as a net:



Addition can be specified with the interaction rules displayed in Figure 2.2. They correspond to the following function definition:

$$add(0, y) = y$$
$$add(s(x), y) = s(add(x, y))$$

Figure 2.3 shows the reduction of a net representing the term $1 + (0 + 1)$.

As we see, interaction rules work in a quite simple and restricted way. Only active pairs can be rewritten, and all auxiliary ports connected to the active pair are preserved. This leads to two key properties of interaction nets, *uniform confluence* and *locality of reductions*.

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

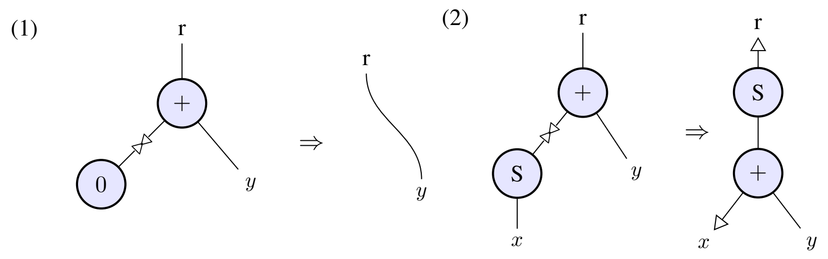


Figure 2.2: Interaction rules for addition

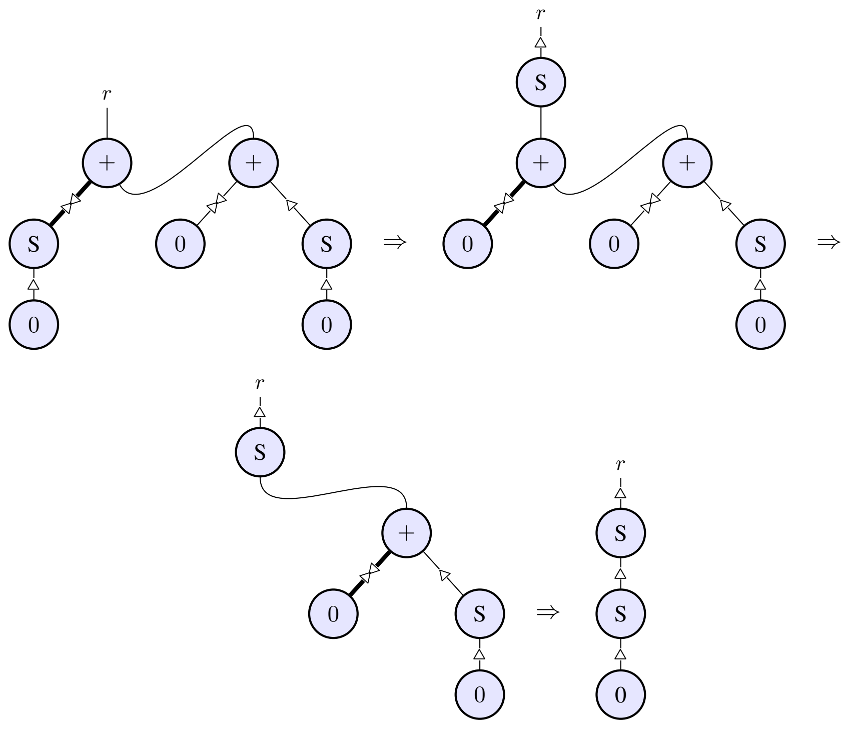


Figure 2.3: Reduction of 1+(0+1) (the active pair reduced in each step is bold).

Proposition 2.2.3 (**Uniform Confluence**, Lafont [30]). *The reduction relation $\Rightarrow$ induced by any INS satisfies the* uniform confluence *property: 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 $P \Rightarrow R \Leftarrow Q$.*

Three properties of interaction nets are sufficient for uniform confluence [32]:

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 agents there is *at most one* rule that can rewrite $S \sim T$. If $S = T$, the rewriting $S \sim T$ must yield the same net as rewriting $T \sim S$ (symmetry).

The last condition is of particular interest if a rule LHS consists of two identical agents, also known as *self-interaction*. In this case, the RHS must be *symmetric* in the sense that the orientation of the active pair is irrelevant (see [32] for details on this condition).

The *linearity* and *binary interaction* properties also ensure *locality of reduction*, which states the aforementioned preservation of interfaces of active pairs: no replacement of an active pair can influence another one. The consequence of these properties is that there is no required order in which active pairs have to be reduced. The result of the computation is always the same, independently of the order of evaluation.

Parallel Evaluation

Due to uniform confluence and locality of reduction, active pairs may be reduced in parallel. Figure 2.4 shows a reduction of our initial example, where all concurrent active pairs are reduced at the same time in every step. Note that the number of concurrent active pairs varies during the course of the reduction, which will be of major importance in Section 5.7.

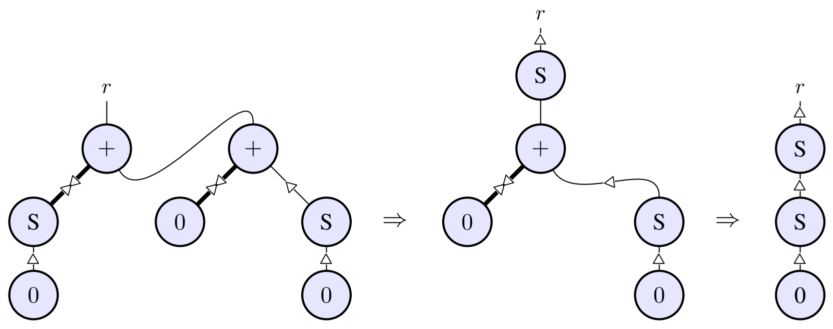


Figure 2.4: Parallel reduction of 1+0+1: In the first step, the net has two active pairs, which can be rewritten at the same time.

Parallel evaluation is an enormous advantage for a programming language based on interaction nets: The current hardware developments show a strong trend towards multi-processor systems. Computers with four CPUs are common nowadays, and the number of processors seems to increase rapidly. Moreover, graphics processing units (GPUs) with hundreds of parallel processors are widely used for compute-intensive applications. With this evolution comes the challenge of using the provided resources efficiently. However, many existing programming languages are tedious to use with respect to concurrency: Computations that may occur in parallel have to be explicitly defined to do so. Precautions have to be taken that concurrent computations do not interfere with each other in unintended ways. In contrast to this, interaction nets provide a way to use parallelism "for free": When defining a program as a net, one does not have to worry about which parts of the algorithm can be done in parallel. One could even imagine the translation of a single-threaded program into the interaction nets paradigm to allow for its parallel evaluation.

Explicit Duplication and Deletion

Another important aspect of interaction nets is that they capture low-level aspects of computation. Duplication, deletion or substitution of values are explicitly performed. Other prominent models such as λ-calculus or term rewriting systems do not cover these aspects. To illustrate this, we give an interaction net encoding of multiplication, similar to the one of addition before. We also introduce an agent ϵ, which removes all other interacting agents, and an agent δ, which duplicates all other agents. Figure 2.5 shows the rules for multiplication, ϵ and δ. When expressed as a function, they would correspond to:

$$\begin{aligned} mult(0, y) &= 0 \\ mult(s(x), y) &= add(y, mult(x, y)) \end{aligned}$$

On the right-hand side of the first equation, y no longer occurs. Its removal is explicitly modeled by the ϵ agent. On the right hand side of the second equation, y is used twice. This duplication is modeled by the δ agent.

Furthermore, interaction nets are efficient. Only nets in normal form can be duplicated, which means that computation is naturally shared (i.e., expressions are only computed once, no matter how often they are used). This has lead to interaction net encodings of the λ-calculus that are more efficient than its implementation in many current functional programming languages. More details and performance results can be found in [39]. We can even use the graphical notation of interaction nets to show properties of programs. In [38], Mackie shows that interaction rules can be used to straightforwardly reason on memory consumption of programs.

Even though interaction nets are very simple, their good properties and modeling capabilities show great potential for a programming language based on this model. Still, the development of such a language stands at the very beginning. To draw an analogy to functional programming again, interaction nets are comparable to the basic λ-calculus, lacking any advanced programming language features or syntactic sugar. The goal of this thesis is to improve this deficiency.

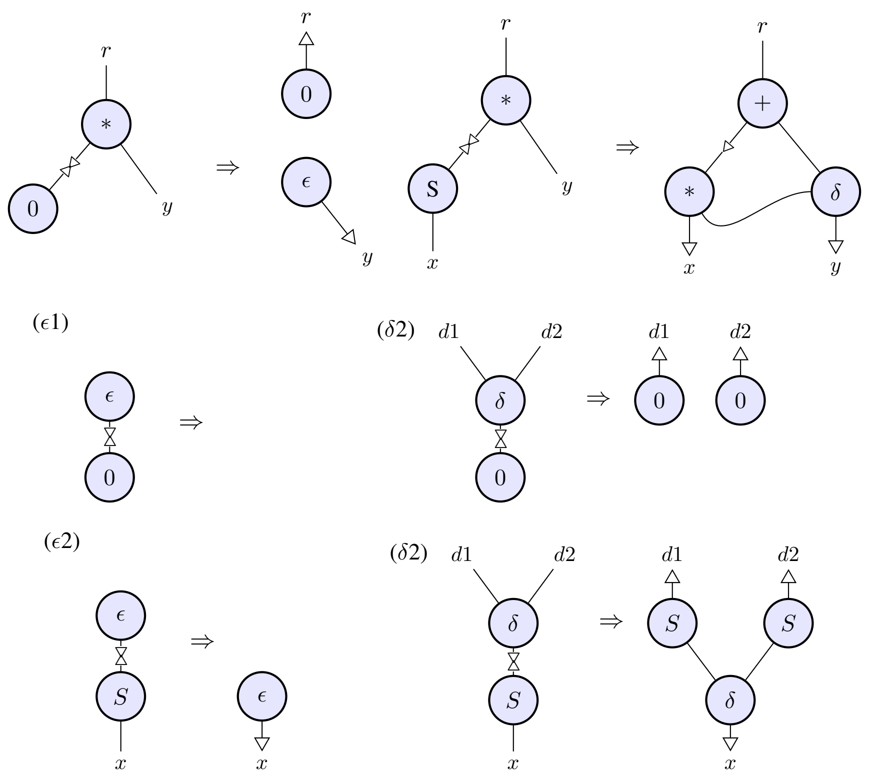


Figure 2.5: Rules for multiplication

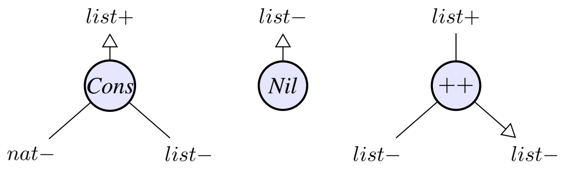
2.2.1 Interaction Nets vs. Term Rewriting Systems

As both INs and TRSs are rewriting systems, one might ask the question of the relation between the two formalisms. Indeed, the initial examples in each paradigm we have given so far seem quite similar. However, the main differences lie in the properties of INs discussed above. First, every INS satisfies the uniform confluence property. Hence, any net can be evaluated in parallel (cf. Figure 2.4. The same does not hold for TRSs. Term rewriting rules are far less constrained and generally might not satisfy any form of confluence. Thus, TRSs model some aspects of computation that INs do not (e.g., non-determinism). On the other hand, TRSs do not model low-level aspects of computation like INs do, leaving duplication and deletion of variables implicit.

TRSs have been extensively studied for decades. There are numerous results on confluence and termination of various subclasses of TRSs. Indeed, it might be interesting to show if certain forms of TRSs correspond to INSs. Then, one could apply the TRS techniques to show termination and other properties to INSs. It turns out that some forms of interaction nets (*non-dependent semi-simple* nets) can be subsumed by certain classes of TRSs [10]. However, general interaction nets cannot be modeled in (first-order) TRSs, as this requires a notion of bound variables. This issue and other relations between INs and TRSs are discussed in detail in [10]. The authors propose a way to model general interaction nets via *combinatory reduction systems*, which incorporate bound variables similarly as in the λ-calculus.

2.3 Type Systems for Interaction Nets

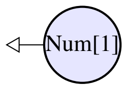
Several type systems have been proposed for interaction nets. In [30], Lafont defines a simple system that assigns a *base type* (int, char, ...) to every port of an agent. Additionally, a *polarity* $(+/-)$ is assigned to each port. Agents may only be connected via ports of the same type, but opposite polarity. Intuitively, polarities divide ports into input and output ports. For example, the list and concatenation agents may be typed as follows:



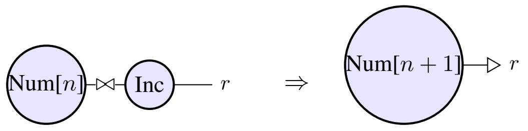
This system is not expressive enough for generic rules and agents that model abstract datatypes (which are a part of monads). For example, there are no type variables, which are essential for expressing monadic data types such as *Maybe a*.

In [8], 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: for example, type assignment of intersection types is undecidable for interaction nets.

A different approach to types in interaction nets is to extend agents and rules with *externally defined programs and data* [13]. For example, instead of a symbolic representation of natural numbers as in Figure 2.2, one could add a natural number as a sort of attachment to a *Num* agent:



We can then define interaction rules that manipulate the external data values when the active pair is rewritten. Consider an agent *Inc* that increments a number:



Of course, more complex external data and programs can be added to INs. As long as the external programs are confluent, the uniform confluence property of INs is preserved [13].

2.4 Lightweight Interaction Calculus

In this section, we recall the main notions of the lightweight interaction calculus [19]. This textual formalism provides a precise semantics of interaction nets and is the basis for our programming language implementations.

2.4.1 The Lightweight Interaction Calculus

The lightweight calculus 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 2.2 corresponds to the following definition for the lightweight calculus.

Definition 2.4.1.1 (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 2.4.1.2 (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.*

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 $\langle\ \bar{t} \mid \alpha(\overline{t_1}) = \beta(\overline{t_2}), \Delta\rangle \xrightarrow{int} \langle\ \bar{t} \mid \Theta', \Delta\rangle$, *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* $\overline{t_1}, \overline{t_2}$.

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

$$\begin{aligned} +(y,r) = S(x) &\longrightarrow +(y,w) = x,\ r = S(w) \\ +(y,r) = Z &\longrightarrow r = y \end{aligned}$$

The following reduction calculates $1+0$*:*

$$\begin{aligned} \langle\ r \mid\ +(r,0) = S(0)\ \rangle \xrightarrow{int} &\quad \langle\ r \mid r = S(x), +(x,0) = 0\ \rangle \\ \xrightarrow{col} &\quad \langle\ S(x) \mid\ +(x,0) = 0\ \rangle \\ \xrightarrow{int} &\quad \langle\ S(x) \mid x = 0\ \rangle \\ \xrightarrow{col} &\quad \langle\ S(0) \mid\ \rangle \end{aligned}$$

Proposition 2.4.1.4 (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 [11], 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 [11]: 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. □

2.5 Pure Languages and Impure Features

We now take a look at functions with *computational side effects*, also referred to as *impure* functions. Extending interaction nets with such functions is one of the main goals of this thesis.

A *pure* (mathematical) function computes its result solely based on the value of its arguments, which is also known as *referential transparency*. This definition is not as trivial as it may look like at first glance: Computer programs frequently incorporate *side effects* such as input/output (I/O) and change of a mutable state. These effects can influence the result of a function independently of its original arguments. Therefore, I/O, state manipulation, exception handling and other important features of imperative programming languages are, while being highly useful and crucial, considered to be *impure*.

To counter this possible source of errors and introduce functions in a more mathematical sense to programming, (declarative) functional languages have been developed. One of the ideas behind functional programming is to promote the purity of functions and to minimize or eliminate side effects. This allows for *equational reasoning* on programs. Equations can be used to construct a formal specification of a program, which has several benefits: On one hand, the implicitly intended behaviour of the program is made explicit. Furthermore, the specification can be used to verify properties of algorithms such as termination or the correctness of their results. Indeed, formal verification of programs can be used to greatly improve the software engineering process.

Interaction nets can be considered a pure language. The reduction of a net is not influenced by anything but its initial state. This forms a solid base for reasoning on properties of programs. For more details on these results, we refer to [34] [9] [10, 35–37]. Yet, apart from the benefits of purity, there are also some disadvantages: Nowadays, software frequently interacts with the real world. Impure functions are considered a vital part of programs, hence pure languages need to incorporate them. This poses problems for pure functions and their good properties, which we will illustrate by an example:

Example 2.5.1. *In functional languages such as Haskell, the evaluation order is* non-strict*, i.e., there is no fixed order in which certain expressions in a function are evaluated. Consider a function in such a non-strict language:*

```
f(x,y) = a - b
  where
  a = x + 1
  b = y - 1
```

When computing the function `f(2,3)`*, it is not specified whether the term* `a=(2+1)` *or* `b=(2-1)` *is evaluated first. Since* `f` *is a pure function, the order of evaluation has no impact on the result.*

In contrast to this, consider a function `g` *which uses the impure function* `readInt` *to read an integer from the input.*

```
g = a - b
  where
  a = readInt
  b = readInt
```

Suppose the input consists of the list of numbers `[2,3]` *and* `readInt` *always takes the first element from that list. Now the order of evaluation effects the result. Depending on whether* `a` *or* `b` *is computed first,* `g` *evaluates to -1 or 1. Thus, referential transparency is lost.*

We see that interaction between computer programs and the real world is a non-trivial topic. Hence, programming languages require an appropriate interface to deal with impure effects. This is especially important for pure languages, as their desired properties are endangered by side effects. Thus, a model of real world interaction that preserves purity of functions and equational reasoning as much as possible is needed. Interaction nets currently lack a formal model for real world extensions. In the following subsection, we discuss a solution approach for this task called *monads*.

2.5.2 Monads

Monads were originally conceived in category theory. Moggi [42] introduced them as a formalism to structure the semantics of programming language features such as state, exceptions or continuations. By adapting monads to the λ-calculus, he created a framework to model these and many more features. Wadler extended Moggi's work by developing a monad model for functional programming languages [50]. This resulted in the implementation of a monad framework in Haskell to realize monadic versions of I/O, state manipulation, array update, exception handling and many more [26].

Essentially, monads provide the means to structure pure functions in order to mimic impure features. By doing so, referential transparency and the possibility for equational reasoning are conserved. To illustrate this, we provide a short formal definition of monads and an example of their use in Haskell. A monad is a triple of an abstract datatype `M a` and two (higher-order) functions operating on this type:

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

`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 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` acts as a left and right neutral element for >>= due to laws (1) and (2). The third law forces a form of associativity property for >>=. As an example, we introduce the *Maybe* monad, which will serve as a running example through the following chapters.

Example 2.5.2.1. *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
```

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 >>=, whereas `Nothing` is returned by >>= without using `f`.

Two aspects of this example cannot be modeled with regular interaction rules. First, the symbol `f` is a variable that represents an arbitrary function. Interaction rules consist of concrete agents only. Second, `Maybe` is a parametrized type with a type variable `a`.

Example 2.5.2.1. ***(monadic I/O)*** *An expression in a pure function* denotes *a value, whereas an I/O command should* perform *some action (e.g., write a character to the output). In Haskell, I/O actions are defined via the monadic type* `IO`*. A type* `IO a` *denotes an action that,* if it is ever performed*, does some I/O and then returns a value of type* `a`*. Simple I/O operations are provided the by following functions, where* `()` *represents a trivial dummy type whose only member is also called* `()`*:*

```
getcIO :: IO Char
putcIO :: Char -> IO ()
```

`getcIO` *is the action which,* when performed*, reads a character from the input and returns that character. Similarly,* `putcIO` *– when performed – writes a character to the output.*

To combine these basic commands to more advanced functions, we use the IO versions of `return` *and >>=:*

```
return :: a -> IO a
>>= :: IO a -> (a -> IO b) -> IO b
```

When performed, `m >>= n` *first performs* `m`*, yielding a result* `IO a`*, and then performs* `n a`*, yielding a result* `IO b`*. We can now define a function* `echo`*: (*`\x -> e` *denotes lambda abstraction,* `\\_ -> e` *discards the argument)*

```
echo :: IO ()
echo = getcIO >>= \c ->
    if (c == eof) then
        return ()
    else
        putcIO c >>= \_ -> echo
```

`>>=` controls the order of evaluation by ensuring that its first argument is performed first and its result is used as input of the second argument. This way, I/O commands are issued in the correct order. Note that other parts of a Haskell program are still evaluated non-strictly: For example, if the argument of `putcIO` is specified by a pure function, it can be evaluated lazily.

In the above example, we put a special emphasis on the semantics of the `IO` type, which denotes a command that does some I/O *if it is ever performed.* This is roughly comparable to a "script" that is performed by executing it. The point at which actions are actually performed is Haskell's *main* function, which acts as the link to the real world. A major advantage of this approach is that it preserves simple equational reasoning: Equal actions can be substituted by one another. This is not possible in side-effect based I/O (see [51] for an example comparison between Haskell and SML).

It is clear that the monad framework in Haskell offers several properties that a real world extension for interaction nets should have. It preserves referential transparency to a large extent and is highly extensible: Haskell uses monads to model exception handling, state manipulation, random generation and many more. Additionally, the programmer can define custom monads for specific purposes. However, the monad model is built on a specific infrastructure, which demands additional tasks when adapting them to interaction nets.

2.6 Logical Foundations of Interaction Nets

While interaction nets can be seen as an instance of an abstract rewriting systen, they have origins in logic. We will only give an overview here, and refer the interested reader to [31] for details. *Linear logic* was introduced by Girard in 1987 [15]. It is a refinement of both classical and intuitionistic logic, of which the latter has a strong correspondence to functional programming languages. In general, (classical) logic deals with the truth of propositions. One characteristic of truth is that it is not consumed when stated. For instance, using the proposition A "I live in Vienna" and $A \to B$ "If I live in Vienna, I live in Europe", one can deduce B "I live in Europe" using classical logic. Furthermore, A is still valid, hence $A, A \to B \vdash A, B$ is a valid judgement.

Consider a different example: Let A be the proposition "Alice has 10 Euros", and B be "Alice has (bought) a pizza (for 10 Euros)". If we want to model the fact that for 10 euros, one can buy a pizza, the rules of classical logic are not sufficient: Indeed, the judgement $A, A \to B \vdash A, B$ is no longer correct in this sense, as buying a pizza would require to spend the 10 euros, hence Alice would have no money left, i.e., A would not be true any more. Linear logic was designed to address this shortcoming. While classical logic allows for the reckless duplication and discarding of propositions, linear logic treats propositions as expendable *resources* that may be used only once in a proof unless explicitly stated otherwise. For example, the classical implication $A \to B$ is replaced by $A \multimap B$, read "consuming A yields B". This corresponds exactly to our example of spending 10 euros to purchase a pizza.

In traditional logic, the grammar of propositions using the common connectives *and*, *or* and *implies* can be specified as follows: (X ranges over propositional variables)

$$A, B := X \mid A \wedge B \mid A \vee B \mid A \to B$$

In linear logic, the conjunction $A \wedge B$ is split in two: We use $A \otimes B$ ("both A and B") to denote the simultaneous availability of both resources and $A \ \& \ B$ ("choose from A and B") the availability of either A or B, where the choice is free. The disjunction $A \vee B$ is replaced by $A \oplus B$, denoting either A or B where one does not have the choice which proposition is available. Together with the linear implication $\multimap$ mentioned above, these connectives specify the grammar of linear propositions:

$$A, B := X \mid A \otimes B \mid A \& B \mid A \oplus B \mid A \multimap B$$

The possibility to duplicate and discard traditional propositions is defined by the logical rules of *contraction* and *weakening*: (presented in natural deduction style)

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \text{Contraction}$$

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}\text{ Weakening}$$

To capture the essence of propositions as expendable resources, linear logic needs to restrict these rules. While it is not advisable to get rid of contraction and weakening completely, their application is limited to specific propositions: If A is a linear proposition that can only be used once, we denote by $!A$ a proposition that is usable an arbitrary number of times in a proof. Hence, only propositions preceded by ! may be duplicated and discarded. For example, we can define the classical implication $A \to B$ (which does not consume A) by $!A \multimap B$.

2.6.1 Proof Nets and the Relation to Interaction Nets

In his work on linear logic [15], Girard also addressed the issue of equality between proofs: In the sequent calculus, proofs for some proposition can differ only in the (sequential) order of inference rules applied. To give a more unique representation of a proof, Girard introduced proof nets, a graphical representation of proofs. Here, we will only consider the *multiplicative fragment* of linear logic, i.e., the $\otimes$ and & connectives. Together with the linear negation $\perp$, the following laws can be established:

$$\begin{aligned} p^{\perp\perp} &= p \\ (A \otimes B)^{\perp} &= A^{\perp} \& B^{\perp} \\ (A \& B)^{\perp} &= A^{\perp} \otimes B^{\perp} \end{aligned}$$

A multiset of formulae is then denoted by the sequent $\vdash A_1, \ldots, A_n$, leading to the four reduction rules $axiom$, cut, $times$ and par:

$$\frac{}{\vdash A, A^{\perp}}\text{ axiom}$$

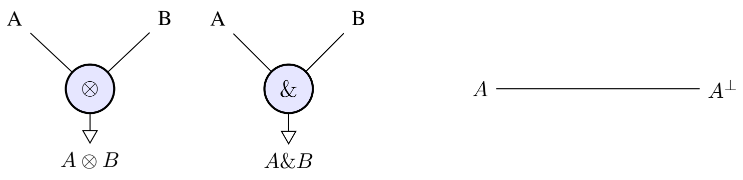
$$\frac{\vdash A, \Gamma \quad \vdash A^{\perp}, \Delta}{\vdash \Gamma, \Delta}\text{ cut}$$

$$\frac{\vdash A, \Gamma \quad \vdash B, \Delta}{\vdash A \otimes B, \Gamma, \Delta}\text{ times}$$

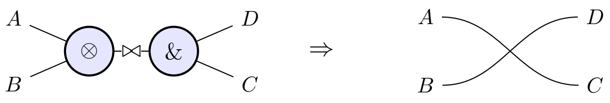
$$\frac{\vdash A, B, \Gamma}{\vdash A \& B, \Gamma}\text{ par}$$

The cut rule is of particular interest: the sequent calculus satisfies the *cut-elimination* property, that is, any proof of some sequent $\vdash \Gamma$ can be reduced to one without the use of cut. Specific reduction rules for cut-elimination in the sequent calculus can be given.

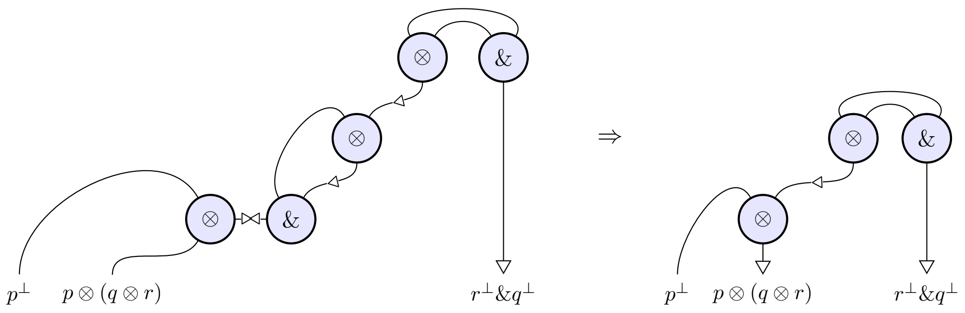
A *proof net* [31] is a graph built from cells labeled with the connectives $\otimes$ and & and wires between them:



These cells have one principal and two auxiliary ports each, and are obviously similar to agents. All ports are assigned types corresponding to a logical formula. This way, proof nets represent proofs of sequents, where the sequent is the *multiset of all interface types* of a net. In addition, proof nets have a single reduction rule:



The interaction of $\otimes$ and $\&$ cells simply removes both cells and connects their corresponding auxiliary ports. Interestingly, this rule corresponds to cut-elimination, i.e., transforming a proof (net) with cuts to a cut-free one. For example, the following nets represent proofs of the sequent $\vdash p^\perp, p \otimes (q \otimes r), r^\perp \& q^\perp$, where the left net reduces to the (cut-free) right one:



Interaction nets have been defined by Lafont [30, 31] as a generalization of proof nets. Instead of just $\otimes$ and $\&$, we have an alphabet Σ of agents (cells) with corresponding arities. Reduction rules are defined between pairs of these agents as introduced in Section 2.2. Computation in INs then corresponds to cut-elimination in proof-nets. This is an example of the *Curry-Howard Isomorphism* which relates programs and proofs (here, nets) and formulas and types (here, interfaces).

2.6.2 Linear Logic and Side Effects

Interestingly, linear logic also relates to our goal of connecting a pure programming language to the real world. A practical example is given by the implementation of I/O in the functional language *Clean* [1].

Example 2.6.2.1. ***(Linear Logic I/O)*** *In the lazy functional language Clean, I/O functionality is realised in an* environment passing style*: All I/O commands accept and return an argument of type* `*World`*, which models the state of the outside world (environment). The important part is that this state of the world has the* "uniqueness" *attribute (indicated by* $\star$*), which is a refinement* *of the constraints of linear logic. An object that is unique may only be used once in a program and never be duplicated. I/O functions interact with the world and return it in an updated state:*

```
putChar :: Char -> *World -> *World
getChar :: *World -> *(Char, *World)
```

`putChar` takes a character and the state of the world and returns an updated state. `getChar` takes a state of the world and returns a pair of the input character and the updated state. Note that `*(Char, *World)` contains 2 uniqueness operators. As mentioned before, the state of the world must be used only once. To ensure this, the pair of the input character and the state must be unique as well. Otherwise, duplication of the state would be possible via duplication of the pair. In contrast to this, the input character alone can be extracted from the pair and used without any restrictions.

Using these basic functions, we can again define `echo`:

```
echo :: *World -> *World
echo c world =
    if (c == eof) then
        world2
    else
        echo world2
    where
        (c,world1) = getChar world
        world2 = putChar c world1
```

While a linear logic type approach to side effects is viable, we have decided to focus on using monads for our side effect extension in interaction nets: As we will show in the later chapters, monads allow for a concise implementation of impure functions that fits well with the style of interaction nets.

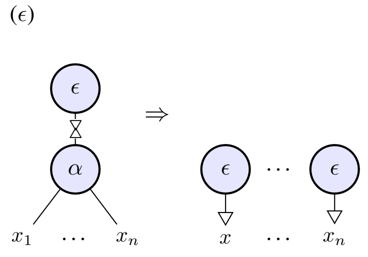
CHAPTER 3

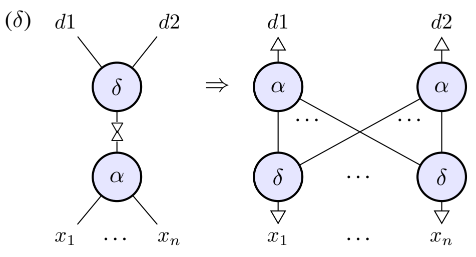
Foundations and Theoretical Results

In this chapter, we present the theoretical results of our work. The main topic is the notion of generic interaction rules, which is based on joint work with Bernhard Gramlich. We first introduce generic rules in the standard graphical setting of INs. We discuss their properties and define a type system that will enable us to define our side effect extension in Chapter 4. We also extend the lightweight interaction calculus with generic rules, and show that the same properties as in the graphical setting hold. Finally, we introduce interaction rules with nested patterns, a conservative extension for more complex rule patterns. We show that nested patterns and generic rules can be fruitfully combined.

3.1 Generic Rules and Imposed Constraints

An ordinary interaction rule matches one specific pair of (distinct) agents only. However, several papers ([8, 9, 34]) 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 we already introduced in Chapter 2. These agents duplicate and erase, respectively, any other agent (including agents without any auxiliary ports):





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 2.2

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 interpreted and restricted in a way that uniform confluence is preserved.

Generic agents can be classified into two kinds. They may either have a fixed arity or have a non-fixed, arbitrary number of auxiliary ports (such as δ or ϵ). We call the latter *variadic* or arbitrary-arity agents. Note that generic agents may only appear in interaction rules, not in concrete instances of nets.

3.1.1 Generic Rules

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. 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. We use INS$_{\mathrm{G}}$ for INS with generic rules.

Definition 3.1.1.1 (**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 3.1.1.2 (**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 $A \bowtie B$.*

Restriction of Self-Interaction Note that generic rules do *not* match active pairs where both agents are the same (e.g., $B \sim B$ for the rule $\alpha \bowtie B$). The reason for this is that some generic rules inherently do not satisfy the “symmetry” condition of Lafont’s *no ambiguity* property (see Section 2.2). However, we can instead add an additional ordinary rule for $B \bowtie B$ to our set of rules, particularly under the constraints defined in the remainder of this section, provided it satisfies the aforementioned symmetry constraint for self-overlaps.

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 3.1.1.3 (**Overlaps**). *Two (distinct) rules in an INS$_{\mathrm{G}}$* overlap *if there exists a single active pair (w.r.t. some INS) which is matched by both rules and can also be rewritten by them.*[1]

[1]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.

Let (Σ, R) be an INS$_{\mathrm{G}}$. 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$_{\mathrm{G}}$ 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 $A \sim B$.*

We now define our constraints for generic rules, first for agents with fixed arity. Afterwards, we extend these constraints to generic agents with arbitrary arity (e.g., δ and ϵ).

We can prevent OG-overlaps by giving priority to ordinary rules. If for an active pair an INS$_{\mathrm{G}}$ has no matching ordinary rule, only then may a generic rule be applied.

Definition 3.1.1.1 (Default Priority Constraint (DPC)). *Let $\mathcal{R} = (\Sigma, R)$ be an INS$_{\mathrm{G}}$. Then $\mathcal{R}$ satisfies the* Default Priority Constraint (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. One can easily see that this completely prevents OG-overlaps. 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.

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

Definition 3.1.1.2 (Generic Rule Constraint (GRC)). *Let $\mathcal{R} = (\Sigma, R)$ be an INS$_{\mathrm{G}}$. R satisfies the* Generic Rule Constraint (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. The combination of GRC and DPC prevents any rule overlaps.

Proposition 3.1.1.3 (Uniform Confluence). *Let $\mathcal{I} = (\Sigma, R)$ be an INS$_{\mathrm{G}}$ that satisfies GRC. Then $\Rightarrow_{R_{DPC}}$ has the uniform confluence property.*

Proof. It is sufficient to show that the *no ambiguity* property of Section 2.2 is preserved by the constraints. 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:

1. Condition (1) holds: In this case, there are no GG-overlaps.

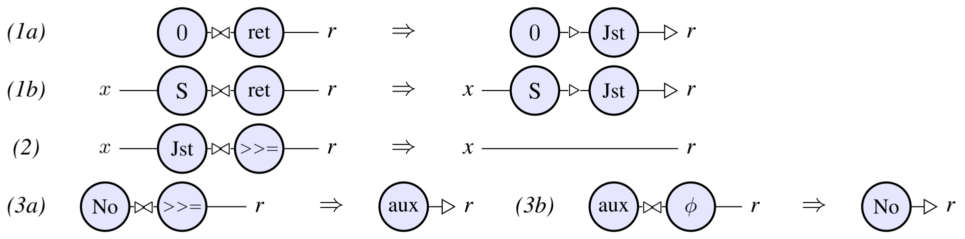
2. Condition (2) holds: There is a rule $A \bowtie B \in O(R)$ that matches the active pair causing a GG-overlap. This overlap is prevented in $\Rightarrow_{R_{DPC}}$, as the ordinary rule has priority over all generic ones.

□

Example 3.1.1.4. *Recall Haskell's* Maybe *monad from Example 2.5.2.1:*

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

The Maybe *monad's* `return` *and* `>>=` *functions for natural numbers can be modeled in interaction nets by the following INS:*



The rule labels correspond to the textual definition of the monadic functions. In rule (3b), we use a generic agent to model the higher-order parameter `f` of `>>=`. We use unary agents for both `return` and `>>=`. This choice has some advantages that we will discuss in detail in Chapter 4.

Even though we employ a generic agent to model higher-order parameters, the above INS is not quite as powerful as Haskell's *Maybe* monad. It is only defined for natural numbers (i.e., the agents S and 0) instead of arbitrary datatypes. This means that `return` should also be defined as a generic rule. In addition, ret should interact with an agent of arbitrary arity, similar to δ and ϵ. Likewise, rule (3b) only defines interaction with a generic agent of arity 1. To improve this issue, we now focus on generic agents with arbitrary arity.

3.1.2 Generic Rules with Variadic Agents

Consider again the δ and ϵ rules. 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).

Definition 3.1.2.1 (Variadic Rule Matching). *Let $r = \alpha \bowtie B$ be a generic rule, where α is of arbitrary arity (denoted by the* dot-notation *"…"). 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.

Variadic Rule Application As indicated by the δ/ϵ rules on page 27, rules with variadic agents may have an arbitrary number of identical agents (or subnets) in their RHS (denoted by "...") . Such a rule $\alpha \bowtie B \Rightarrow N$ is applied to an active pair $A \sim B$ as follows: let $n = arity(A)$. $A \sim B$ is replaced by a net N', such that N' contains n copies of all agents, wires and ports in N *that are marked with the dot-notation* and one copy of the remaining parts of N (which are not marked with the dot-notation). For example, consider the δ-rule on page 27: the δ agents, the ports x_i and all wires between δ and α agents are copied n times. Both α agents and the ports $d1, d2$ only appear once in N'.

The constraints and properties of fixed-arity generic rules can be extended to the arbitrary arity case. For this, we define the notion of *arity unfolding*.

Definition 3.1.2.2 (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).

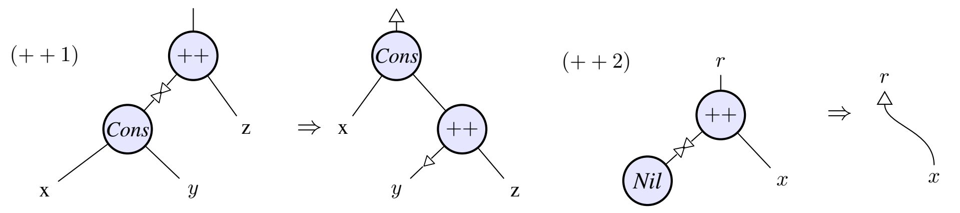
Theorem 3.1.2.3. *Let $\mathcal{I} = (\Sigma, R)$ be an INS$_{\mathrm{G}}$, where R contains at least one generic rule with a variadic agent. If $AU(R)$ satisfies GRC, then $\Rightarrow_{R_{DPC}}$ satisfies the* no ambiguity *property.*

Proof. If $AU(R)$ satisfies GRC, then it has no GG-overlaps, except those that are prevented by DPC. Since $N \Rightarrow_R M$ if and only if $N \Rightarrow_{AU(R)} M$, R has no additional GG-overlaps either. Hence, $\Rightarrow_{R_{DPC}}$ has no overlaps. □

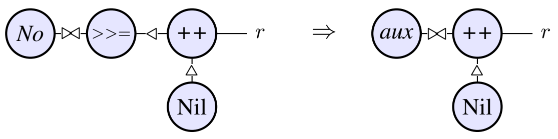
3.1.3 Non-Uniform Port Handling

In the RHS of a variadic rule, all (arbitrarily many) ports of the generic agent are handled in the same, *uniform* way. However, we can define generic rules where a few selected ports of a generic agent receive a different, non-uniform treatment and the remaining, arbitrarily many ports are handled uniformly. Such rules can be used to support *curried functions* or *partial applications* (e.g., `(1+)` in Haskell)

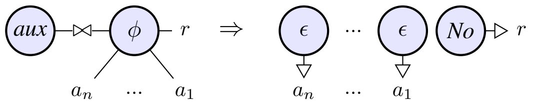
We now give an example for non-uniform port handling. First consider the following INS that allows us to concatenate lists of natural numbers: $\mathcal{R} = (\Sigma, R)$ where $\Sigma = \{0^0, S^1, Cons^2, Nil^0, ++^2\}$ and R is defined as follows:



It is straightforward to see a net consisting of the agents $++$ and *Nil* as the interaction net equivalent of the partially evaluated function `(++ Nil)`. Unfortunately, the interaction rules of the *Maybe* monad from Example 3.1.1.4 are not applicable here. Consider the following reduction:



Even though the second net has an active pair, it cannot be reduced any further. The ++ 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 $++$ in the previous example. This updated generic rule is applicable to the net above: rule matching for the non-uniform case works almost identically to Definition 3.1.2.1. However, the respective agent of the active pair needs to have at least the number of non-uniformly handled auxiliary ports.

Definition 3.1.3.1 (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. r* matches *an active pair $A \sim B$ 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 3.1.2.3 and its result on uniform confluence.

3.2 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, matching any agent with the arity of the generic agent. Naturally, this problem is solved by a type system for interaction nets. In this section, we will define a type system that is suitable for expressing monadic rules in interaction nets.

3.2.1 Our Typing Approach

The main goal of our type system is to restrict the matching power of generic rules. We decided to keep the system as simple as possible. However, our system needs to be expressive enough to model types of the monadic operators:

```
return :: a -> M a
>>= :: 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 [8] offers arrow types for ports, we feel that this system is overly complex for our needs. 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 3.2.1.1 (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* $s \in S, p \in P, t_i \in PT (1 \leq i \leq n)$*,* $arity(s) = n$

All types of an agent's ports form its *environment*. This notion was already introduced in [8]. However, we define it in a slightly different way, ordering the types by their port positions, starting counter-clockwise from the principal port.

Definition 3.2.1.2 (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, $\varepsilon(Cons) = \{list(\tau)^+, \tau^-, list(\tau)^-\}$. Within the environment, multiple occurrences of the same variable (here: τ) refer to the same type. I.e., the environment is the *scope* of port 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 3.2.1.3 (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 $\{t_1 \approx r_1, \ldots, t_n \approx r_n\}$.*

A well-typed net can be seen as the equivalent of a well-typed program. The definition ensures that the environments of each individual agent remain consistent: after applying the substitution resulting from the substitution problem above, the same variables within the scope of an environment are replaced by the same type.

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

Definition 3.2.1.4 (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. Think of a rule LHS where one auxiliary port of each active agent needs to have the same variable type as the other. Since the scope of the variable is the environment of a single agent, we need additional information to declare two port type variables of different agents as equal. This is the purpose of S, as is shown in the following example:

Example 3.2.1.5. *Consider rule (2) of the Maybe monad in Example 3.1.1.4:*
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, $RT((2)) = (\varepsilon(Jst), \varepsilon(>>=), \{\tau \mapsto \rho\})$. 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.

Definition 3.2.1.6 (Well-Typed Rules). *A rule is* well-typed *if both nets of the LHS and RHS are well-typed and 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$_{\mathrm{GT}}$ for an INS$_{\mathrm{G}}$ with typed rules.

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 3.2.1.7 (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 r* matches *$A \sim B$ w.r.t. Definition 3.1.1.2 and there exists a type substitution σ s.t. $\sigma(S(\varepsilon(\alpha))) = \varepsilon(A)$, where S is the substitution of $RT(r)$.*

3.2.2 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 3.2.2.1 (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.

Intuitively, $*$ may be replaced by any number of port types. 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 3.2.2.2 (Typed Rule Matching with Variadic Agents). *Let $r = \alpha \bowtie B$, where α is a variadic agent. Let $\varepsilon(\alpha) = (t^p, *, {t_1}^{q_1}, \dots, {t_m}^{q_m})$, 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.*
  $\varepsilon(\alpha') = (t^p, {x_1}^{p_1}, \dots, {x_n}^{p_n}, {t_1}^{q_1}, \dots, {t_m}^{q_m})$, 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 3.2.1.7.*

The simplicity of the type system makes it quite easy to handle. For example, well-typedness of nets is decidable in linear time.

3.2.3 Properties of the Type System

The presented type system combines well with the generic rule constraints of Section 3.1: 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 3.2.3.1. *Let N be a typed net without generic agents. Given the environments of all agents, well-typedness of N can be decided in linear time (i.e., linear in the number of agents and ports of the net).*

Proof. Both properties of Definition 3.2.1.3 can be shown in linear time. In particular, the second property is a unification problem which can be solved in linear time [5]. By Definition 3.2.1.3, two properties need to hold:

- all connected ports are of opposite polarity.

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

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 [5]. Therefore, the overall time complexity of deciding well-typedness of a net is linear in the number of agents and ports of the net. □

3.3 Generic Rules: Summary

We have defined generic rules for interaction nets, which are substantially more powerful than ordinary interaction rules. Generic rules allow us to conveniently express higher-order functions in interaction nets, in particular monads, which we will show in Chapter 4. Generic rules are a conservative extension: Under appropriate constraints (DPC and GRC), the uniform confluence property of interaction nets is preserved.

The matching power of generic rules can be further restricted by a type system. We defined a system that is expressive enough for generic rules and monads, but still offers nice properties due to its simplicity.

3.4 Generic Rules for the Lightweight Calculus

In the previous subsections, we introduced generic rules in the graphical setting of interaction nets. Now, we extend the lightweight interaction calculus (see Section 2.4.1) in order to express the semantics of generic rules. In particular, this will provide us with a precise definition of variadic rule matching and application.

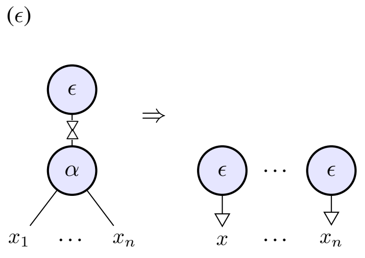
3.4.1 Fixed and Variadic Generic Rules

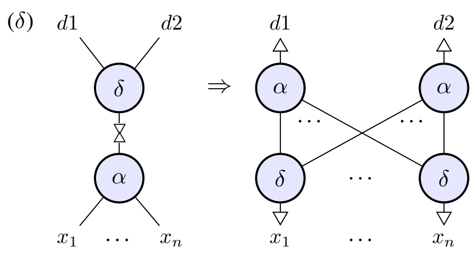
As we discussed in the previous part of this chapter, we distinguish two types of generic agents and rules based on the arity of the agent:

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.

Recall the rules for 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.

3.4.2 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 3.4.2.1 (Generic Interaction). $\langle\ \bar{t} \mid \alpha(\overline{t_1}) = \beta(\overline{t_2}), \Delta\rangle \xrightarrow{int} \langle\ \bar{t} \mid \Theta', \Delta\rangle$, *where* $\alpha(\overline{x}) = \beta(\overline{y}) \longrightarrow \Theta\ \in$ R *or* $\alpha(\overline{x}) = \phi(\overline{y}) \longrightarrow \Theta\ \in$ 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 3.4.4.

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.

3.4.3 Generic Rule Constraints

Just as in the graphical setting, 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 2.4.1.4, *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 [25], 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 (see Definition 3.1.1.1).

Definition 3.4.3.1. *We define the following constraints for the generic lightweight calculus:*

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 $\alpha(\overline{x}) = \beta(\overline{y}) \longrightarrow \Theta \ \notin$ 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 3.4.3.2. *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 3.4.2.1). 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 Proposition 3.1.1.3, we can now show uniform confluence of the lightweight calculus with generic rules.

Proposition 3.4.3.3 (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 2.4.1.4: 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 3.4.3.2 shows that the generic rule constraints prevent any ambiguity that might arise from the application of the $\xrightarrow{int}$ rule. □

3.4.4 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 (cf. Section 3.4.1):

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' stands for 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 variadic agent into account when replacing an equation.

Definition 3.4.4.1 (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 3.4.4.2. *The variadic rule for deletion via the agent* ϵ *can be defined as follows:* $\epsilon = \phi([x]) \quad\longrightarrow\quad \epsilon = x'$*. Applying the rule to the equation* $\epsilon = A(u, v, t)$ *yields* $\{\epsilon = u, \epsilon = v, \epsilon = t\}$*: 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 3.4.4.3. *The variadic rule for duplication via the agent* δ *can be defined as follows:* $\delta(d_1, d_2) = \phi([x]) \quad\longrightarrow\quad d_1 = \phi([y]),\ d_2 = \phi([z]),\ x' = \delta(y', z')$*. 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. Variadic ranges and names formalize the mechanics expressed by the graphical dot notation.

3.4.5 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* (see Section 3.1.3). 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 3.4.4.1 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 Example 2.5.2.1.

Example 3.4.5.1. *The* Maybe *monad in Haskell 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 3.4.4):

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

The rules are labeled in correspondence with 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 3.4.5.2. *Consider a function* `pick`*, which picks the nth element of a list or returns* `Nothing` *if that element does not exist:*

```
pick :: [a] -> Int -> 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):

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

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

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

3.5 Generic Rules and the Interaction Calculus: Summary

In the previous section, we have defined generic rules in the lightweight calculus. This is a worthwhile addition to our investigation in the graphical setting for two reasons. First, the lightweight calculus provides an alternative, precise semantics of generic rules. In particular, we provide a semantics of variadic rules that concretizes the intuitive dot-notation of the graphical interaction rules. Second, the lightweight calculus is the foundation of the interaction nets based programming language *inets*, which supports generic rules (we will discuss *inets* in detail in Chapter 5).

3.6 Interaction rules with nested patterns - INP

The definition of interaction rules constrains pattern matching to exactly two agents at a time. Consequently, we have to introduce auxiliary agents and rules to perform deep pattern matching. Despite their increased matching power, even generic rules are subject to this restriction.

In this section, we take a step towards developing a richer language for interaction nets which facilitates nested pattern matching. Consider the following definition of a function that computes the last element of a list:

```
last (x:[]) = x
last (x:(y:ys)) = last (y:ys)
```

The definitions above contain three non-variable symbols on each LHS: `last`, the list constructor `:` and the empty list `[]` or a second `:`, respectively. As the LHS of an ordinary interaction rule consists of exactly two symbols, additional auxiliary rules have to be introduced, as shown in Figure 3.1. This would correspond to the following textual definition:

```
last (x:xs) = aux x xs
aux x [] = x
aux x (y:ys) = last (y:ys)
```

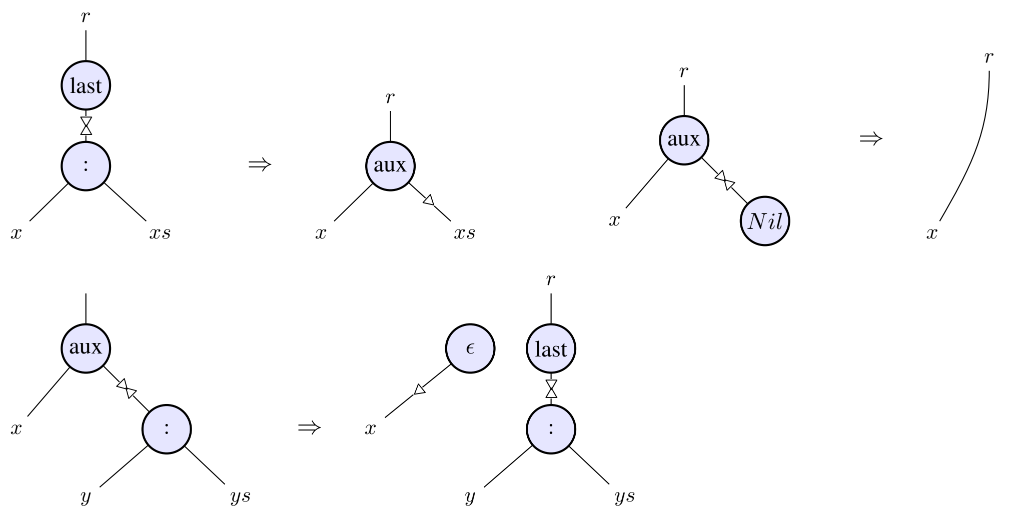


Figure 3.1: computing the last element of a list with auxiliary rules

An interaction rule may contain a *nested active pair* with more than two agents on it's LHS.

Definition 3.6.1 (Nested Active Pair). *A* nested active pair *is defined inductively as follows:*

- *Every active pair in ordinary interaction rules (ORN) is a nested active pair e.g.* $P = \langle \alpha(x_1, \ldots, x_n) \bowtie \beta(y_1, \ldots, y_m) \rangle$
- *A net obtained as a result of connecting the principal port of some agent γ to a free port y_j in a nested active pair P is also a nested active pair e.g. $\langle P, y_j \sim \gamma(z_1, \ldots, z_l) \rangle$*

A nested interaction rule matches *a nested active pair that is identical to its LHS.*

As an example, Figure 3.2 gives a set of INP rules that will compute the last element of a list. In this Figure both rules contain a nested active pair on the LHS. The (non interacting) agents *Nil* and *:* on the LHS of the rules are nested agents. In comparison, the respective set of ORN rules is given in Figure 3.1.

The framework of interaction rules with nested active pairs is called INP (**I**nteraction nets with **N**ested **P**atterns). Rules in INP may overlap due to the more complex LHS patterns. To ensure that INP preserves the uniform confluence property of interaction nets, rules in INP must satisfy the following constraints:

Definition 3.6.2 (Sequential set property [20]**).** *A set of nested active pairs N is* sequential *if and only if, when $\langle P, y_j - \gamma(z_1, \ldots, z_n) \rangle \in N$, then*

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

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

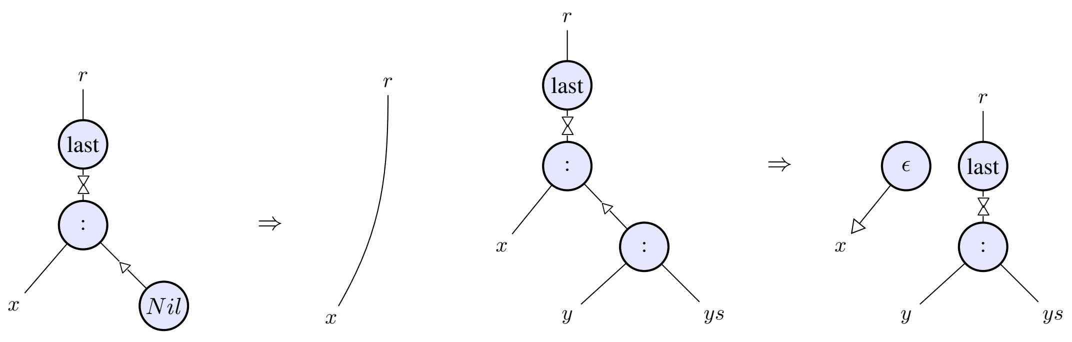


Figure 3.2: example INP rules to compute the last element of a list

Definition 3.6.3 (Well-Formedness [20]**).** *A set of rules R is* well-formed *if and only if*

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

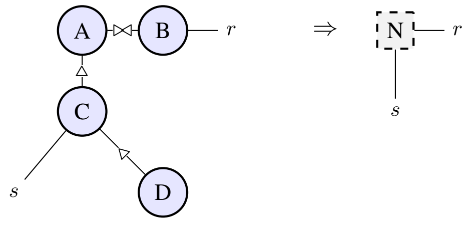
Well-formedness of a set of rules is sufficient for strong confluence: [2]

Proposition 3.6.4 ([20]**).** *If a set of rules R in INP is well-formed, then the reduction relation induced by R is strongly confluent.*

Intuitively, the Sequentiality property avoids overlaps between rules: a set of INP rules containing nested patterns that violate condition 2 of Definition 3.6.2 can give rise to critical pairs, which potentially destroys the uniform confluence property of interaction nets. Note that the definition of Sequentiality allows a nested active pair to be a subnet of another nested active pair (in the same sequential set) which may also give rise to critical pairs. Definition 3.6.3 ensures that there is at most one nested active pair in any given set of rules.

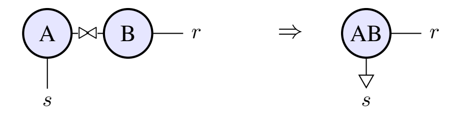
3.6.1 Translating Nested Patterns to Ordinary Rules

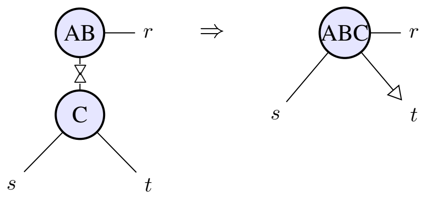
Any well-formed set of INP rules can be translated to a set of ORN rules [20]. We discuss a concrete algorithm and its implementation in Chapter 5. The idea is to introduce auxiliary agents and rules, one of each for every nested agent. As an example, consider the following INP rule:

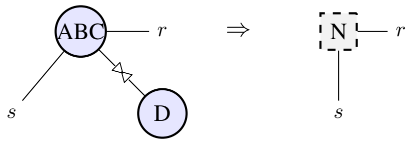


[2] More precisely, by strong confluence we here mean the property: Whenever $M \Leftarrow N \Rightarrow P$ with $M \neq P$, then there exists a net Q s.t. $M \Rightarrow Q \Leftarrow P$

This INP rule is transformed into a set of three ordinary interaction rules:







Auxiliary agents AB, ABC are added to model the (sequential) matching of the nested agents. This translation is important for our implementation of nested patterns in the interaction nets based language *inets* (again, see Chapter 5).

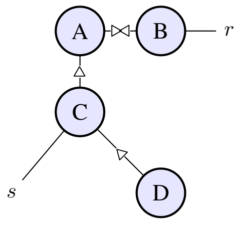
3.7 Combining Generic Rules and Nested Patterns

In this section, we show that nested patterns and generic rules can be combined to allow for even more powerful interaction rules.

A nested active pair can be seen as a tree, where the pair of active agents forms the root. Nested agents may be connected to this pair or other nested agents' auxiliary ports, forming a *tree-like structure*.

Definition 3.7.1 (Nested Pattern Tree). *Let r be an INP rule. The* nested pattern tree *of r is a tree where the root consists of the active pair of r and each nested agent is a node (connected to the tree through its principal port).*

As an example, the following nested pattern can be seen as a tree with the pair $A \bowtie B$ as the root node and D as a leaf node:



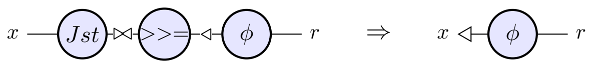
To preserve uniform confluence, we introduced constraints for a given set R of INP rules in Definition 3.6.3. Intuitively they are:

1. No LHS in R may be a *subnet* of (or equal to) another.

2. No two LHSes N, M may *overlap*, in the sense that there is no concrete net P that matches both N and M. (aka *sequential set* property)

3.7.2 Generic Nested Rule Patterns (GINP)

The motivation for nested patterns with generic agents comes from the definition of higher-order functions, in particular monads: the `bind (>>=)` rule (cf. Section 2.5.2) has two arguments (`M >>= f`), where `f` is an arbitrary function. In Example 3.1.1.4, we used two rule with an auxiliary agent to model >>=. The corresponding generic interaction rule would be



where ϕ is the generic agent. In this case, ϕ represents the nested part of the LHS, it is not part of the active pair. For simplicity, we will restrict ourselves to nested patterns with one generic agent only, though our results can be extended to multiple generic agents in a straightforward way.

Definition 3.7.2.1 (**Generic Nested Rules**). *A* generic nested interaction rule *(GINP rule for short) is an interaction rule with a nested active pair on its LHS. The nested active pair contains exactly one generic agent, with the following constraints:*

- *a variadic agent without non-uniformly handled ports may only appear at the leaves of nested pattern tree.*
- *a variadic agent with non-uniformly handled ports may appear at any position, but no agents may be connected to its uniformly handled ports.*
- *a fixed generic agent may appear at any position in the nested pattern.*

The constraints to variadic agents restrict the matching power of generic nested patterns. Without them, a single pattern with a variadic agent could match a variety of nets that do not share the same nested tree structure.

Definition 3.7.2.2 (**GINP Rule Matching**). *A GINP rule r matches a nested active pair α if its LHS and α are the same except for r's generic agent, and the agent in α corresponding to the generic agent has a suitable number of ports w.r.t. Definitions 3.1.1.2, 3.1.2.1 and 3.1.3.1.*

We can extend the notion of GINP matching to our type system of Section 3.2. First, we slightly adapt our definition of rule types:

Definition 3.7.2.3 (**GINP Rule Type**). *Let r be an GINP rule whose LHS consists of the agents $A_1, \ldots, A_n$ and the type variables of all agents' ports are disjoint. The* rule type $RT(r)$ *is a tuple $(\varepsilon(A_1), \ldots, \varepsilon(A_n), S)$, where S is a* type substitution $\{\tau_1 \mapsto t_1, \ldots, \tau_n \mapsto t_n\}$ *consisting of types of the nested agents' environments.*
The rule r is well-typed *if both nets of the LHS and RHS are well-typed and the types of all free ports are the same in the LHS and RHS.*

Definition 3.7.2.4 (GINP Typed Rule Matching). *Let r be a well-typed GINP rule where α is a generic agent. Let N be a well-typed net containing a nested active pair α. We say that r* matches *α if r matches α w.r.t. Definition 3.7.2.2 and there exists a type substitution σ s.t. $\sigma(S(\varepsilon(\phi))) = \varepsilon(A)$, where S is the substitution of $RT(r)$, ϕ is the generic agent in r's LHS and A is the agent in α corresponding to ϕ.*

Intuitively, GINP rule matching works analogously to generic rule matching without nested patterns. We simply extend our previous definitions to account for the nested pattern.

3.7.3 GINP Constraints

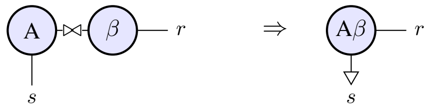
Interestingly, for the purpose of preserving uniform confluence, it is not needed to distinguish whether the generic agent is part of the active pair or the nested pattern. However, this distinction will be important for the transformation of INP to ORN rules (see next subsection).

The extension of the INP constraints (see Section 3.6) is straightforward: it is merely needed to interpret equality between nested patterns in the sense of *generic rule overlaps*. We adapt the notion of a well-formed set of rules in Definition 3.6.3 by changing the subnet constraint: A net P is a generic subnet of Q if we can replace all generic agents in P and Q by ordinary agents of suitable arity such that the resulting nets have the subnet relation. The *sequential set* constraint Definition 3.6.3 does not need to be changed, since it is about overlapping tree structures of nested patterns and not about concrete agents.

The generic rule constraints DPC and GRC can easily be adapted to GINP rules. Both Definition 3.1.1.1 and 3.1.1.2 (and hence, Proposition 3.1.1.3) hold if we simply replace generic rules by GINP rules and ordinary rules by INP rules. Thus, uniform confluence is preserved in the presence of GINP rules.

3.7.4 Extending the INP to ORN Translation

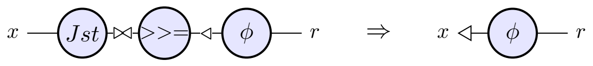
Unfortunately, potential problems arise during the translation of GINP rules to ordinary (generic) rules: auxiliary agents are concrete agents and cannot carry information about the current (concrete) instance of a generic agent. However, this information needs to be propagated through all auxiliary rules, as the generic agent may appear in the final RHS net N. Consider the previous example. Suppose we replace B by the generic agent β, then the first auxiliary rule could look like this:



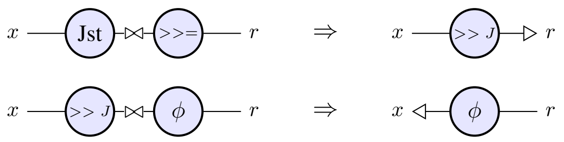
The LHS is a standard non-nested generic rule. However, the agent Aβ in the RHS is somehow *neither concrete nor generic*: it is not generic, as it contains information about the concrete agent A. At the same time, its not supposed to be concrete as it contains β, which represents any agent (of suitable arity). This is an issue for the final rule of the translation, let us call it $A\beta C \bowtie D \Rightarrow N$. N may contain β, but there is no information about the concrete instantiation of β when the rule is applied. Of course, one could add rules for every concrete agent, $AAC \bowtie D$, $ABC \bowtie D$, $ACC \bowtie D$,..., but that is hardly a good solution. Therefore, the transformation needs to be extended in order to propagate information about generic agents (or rather, their concrete instances).

3.7.5 An Unproblematic Case

Interestingly, the original transformation may still work in the following case. If the generic agent is the *last nested agent to be matched*, then it does not need to be propagated through auxiliary rules. Interaction of the final nested agent and a previous auxiliary agent does not yield another auxiliary agent, but the original RHS net (as in the rule $ABC \bowtie D$ above). For example, the nested generic rule



translates to:



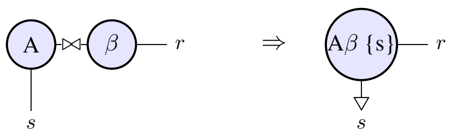
The second rule is a non-nested generic rule. Of course, this translation only works if the single generic agent is at one of the **leaves of the nested pattern tree** (i.e., it is matched last).

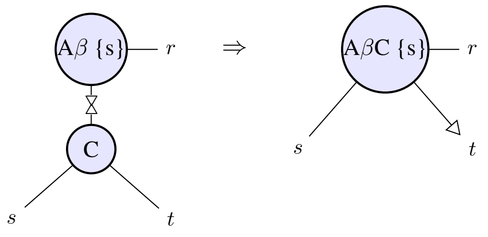
Proposition 3.7.5.1. *Let r be a GINP rule that contains one generic agent at a leaf of the nested pattern tree of r's RHS. Then, the nested pattern transformation of [20] can be applied if the generic agent is translated last.*

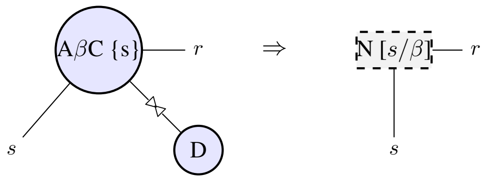
3.7.6 Extending the Translation

If the generic agent is part of the active pair (or nested, but not at a leaf of the nested pattern tree), then the translation fails. It is clear that auxiliary RHSes need to carry additional information: the name of the concrete instance of the generic agent, and possibly the arity in the variadic case. This could be achieved by adding external datatypes to auxiliary RHSes.

Agents with externally defined datatypes [13, 38] (see Section 2.3) hold values of some type in addition to their symbol. Auxiliary agents could hold the symbol of the concrete instance of the generic agent as a string value. In the final RHS N, the generic agent could be renamed using this string value. For example, the rule above could be translated as follows, where s is the external data representing the symbol of the concrete instance of the generic agent and $N[s/\beta]$ denotes substitution of β agents by s agents.







This approach is interesting for several reasons:

- There are theoretical results on agents with external datatypes, e.g., preservation of uniform confluence [13].
- It would combine two previously unconnected features of INs, nested patterns and external datatypes.
- External datatypes are implemented in *inets*, hence the implementation of the translation seems viable.

3.7.7 Extension to Multiple Generic Agents

After a more detailed discussion of the single generic agent case, it seems likely that the case of multiple generic agents in a nested pattern can be also be solved in this way (at least for the fixed arity case). For example, we can extend the "external datatypes" approach by having a *map of generic agents and concrete instance symbols* as external data, which is built up during the application of the auxiliary rules.

If the nested pattern contains more than one generic agent, it still seems straightforward to ensure uniform confluence with the above constraints (a simple extension of generic rule matching is needed).

3.7.8 Generic Nested Patterns: Summary

Combining nested patterns with generic agents is generally unproblematic. The nested pattern constraints can be extended straightforwardly, preserving uniform confluence. There are more subtleties than at first glance, though. Generic agents can be at different positions in the nested pattern tree, and there can be more than one. The original transformation to ORN rules does **not** work in the presence of generic agents and needs to be extended. This is an important issue, as the *inets* implementation of nested patterns uses this transformation. The GINP to ORN translation can be extended with externally defined datatypes, which propagate additional information about the generic agents through the auxiliary rules. This yields a useful combination of *three different interaction nets extensions*: generic rules, nested patterns and external datatypes.

CHAPTER 4

Applications

In this chapter, we show how our theoretical results can be used to realize side effects in INs. Generic typed rules are used to realize monads as INSes. We give examples for different monads and show their correctness.

4.1 Monads in Interaction Nets

Our theoretical efforts of extending interaction nets were aimed at one main purpose: to handle side effects directly in interaction nets while preserving the overall purity of IN computation as well as uniform confluence. We do this by adapting monads - which we introduced in Section 2.5.2 - to interaction nets. Recall that a monad (in the sense of functional programming) is a triple of a parametric datatype `M` and two functions:

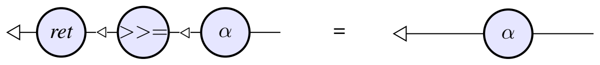
```
return      :: a -> M a
>>= (bind) :: M a -> (a -> M b) -> M b
```

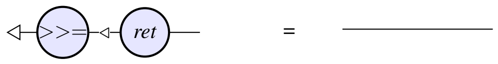
These three objects need to satisfy some constraints expressed by the following three laws:

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

Hence, an interaction net version of a monad needs to feature the three mentioned objects as well as constraints between these objects. Naturally, `return` and `>>=` can be represented as agents that interact with agents of type `M a`. We represent the monad laws as “observational” equalities between nets:

Definition 4.1.1 (**Interaction Net Monad**). *An* interaction net monad *is an INS whose signature contains two unary agents* $>>=$ *and* ret*. The rules of the INS need to satisfy the following* equalities*:*

(1) 

(2) 

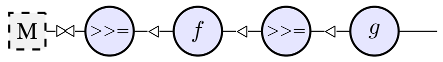
The agent α corresponds to an arbitrary agent, similar to the arbitrary function `f` *in the textual definition* `(1)`*.* Equality *is interpreted as* observational equivalence *[12]: If arbitrary nets are connected to the free ports of both nets of an equation (enabling reduction), then they can be reduced to a common successor.*

The equalities (1) and (2) correspond to the first two monad laws. Equivalence can be shown by giving reduction sequences of nets that yield the aforementioned common successors.

While $>>=$ may be modeled differently (e.g., with more auxiliary ports), our approach captures the essence of monadic side effect handling in a natural and intuitive way. In addition, the third monad law automatically holds due to this representation:

Proposition 4.1.2. *If an agent with only one auxiliary port is used to encode $>>=$ in interaction nets, then the monad law (3)* `(m >>= f) >>= g = m >>= (\x -> (f x >>= g))` *holds.*

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



Therefore, it is trivially true. □

4.2 Application: Monads in Interaction Nets

We now present interaction nets versions of several monads, starting with a revised version of the *Maybe* monad from Example 3.1.1.4. All of these INSs in this chapter are monads in the sense of Definition 4.1.1. This can be shown by a reduction of nets, such that both sides of each equation have a common reduct (similar as in [24]).

4.2.1 The Maybe Monad Revisited

Recall Haskell's *Maybe* monad from Section 2.5.2:

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

Using generic rules, we can express the interaction rules of the *Maybe* monad in a way that closely models Haskell's definition. 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 4.1.

The rule labels correspond to the lines of the definition in Example 2.5.2.1. Lines (2) and (3) are split into two rules each, for two reasons. First, this is due to the standard restriction of two agents per interaction rule LHS. Second, the auxiliary agents $>> J$ and $>> N$ interact with the

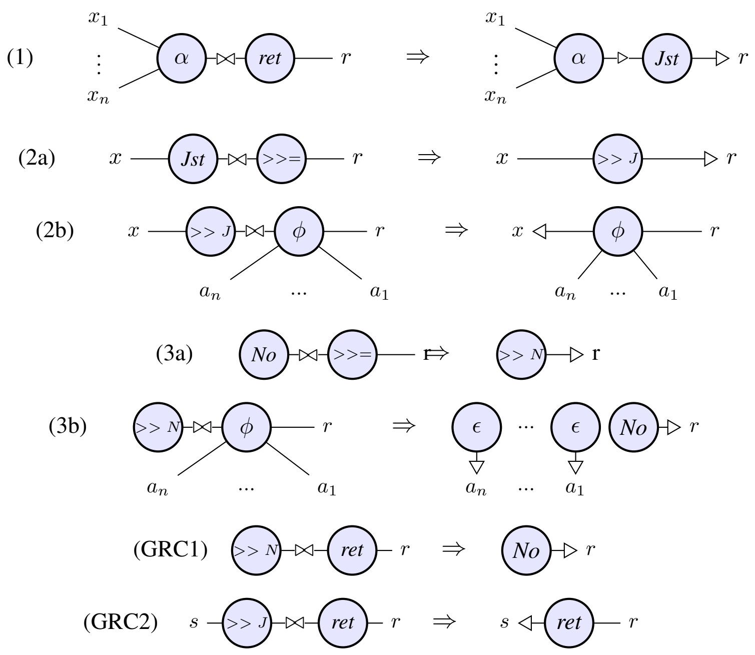


Figure 4.1: The *Maybe* IN monad

generic agent ϕ and thus ensure its correct type. To ensure that the set of rules satisfies the GRC, we add the auxiliary rules (GRC1) and (GRC2).

Proposition 4.2.1. *$\Rightarrow_{M_{DPC}}$ 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 3.1.2.3 $\Rightarrow_{M_{DPC}}$ satisfies the uniform confluence property. □

We assign the environments in Figure 4.2 to the agents of the *Maybe* monad, where $maybe$ is a sort of arity 1 and τ, ρ are type variables. The rule types are given in the right column.

Recall that the scope of port type variables is the agent's environment. We add subscripts to the variables in the rule types 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. Note that in rule type $RT(2)$, the type substitution S requires that both auxiliary ports of the active agents have the same type.

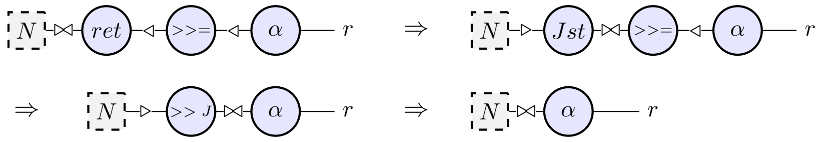
We see that the rule set of the *Maybe* monad is just as expressive as its textual definition in Haskell: it allows any agent as a basic value of the corresponding monad type. Thanks to non-uniform port handling of generic rules, any agent with matching port types can be used as second argument of *bind*.

$$\begin{aligned}
\varepsilon(Jst) &= \{maybe(\tau)^+, \tau^-\} & RT(1) &= \{\varepsilon(\alpha), \varepsilon(ret), \{\}\}\\
\varepsilon(No) &= \{maybe(\tau)^+\} & RT(2a) &= \{\varepsilon(Jst), \varepsilon(>>=), \{\tau_1 \mapsto \tau_2\}\\
\varepsilon(ret) &= \{\tau^-, maybe(\tau)^+\} & RT(2b) &= \{\varepsilon(>>\ J), \varepsilon(\phi), \{\tau_1 \mapsto \tau_2\}\}\\
\varepsilon(>>=) &= \{maybe(\tau)^-, (\tau)^+\} & RT(3a) &= \{\varepsilon(No), \varepsilon(>>=), \{\}\}\\
\varepsilon(>>\ J) &= \{\tau^+, \tau^-\} & RT(3b) &= \{\varepsilon(aux), \varepsilon(\phi), \{\}\}\\
\varepsilon(>>\ N) = \{\ \tau^+\},\ \ \varepsilon(\alpha) &= \{\tau^+, *\} & RT(GRC1) &= \{\varepsilon(>>\ N), \varepsilon(ret), \{\}\}\\
\varepsilon(\phi) &= \{\tau^-, *, maybe(\rho)^+\} & RT(GRC2) &= \{\varepsilon(>>\ J), \varepsilon(ret), \{\}\}
\end{aligned}$$

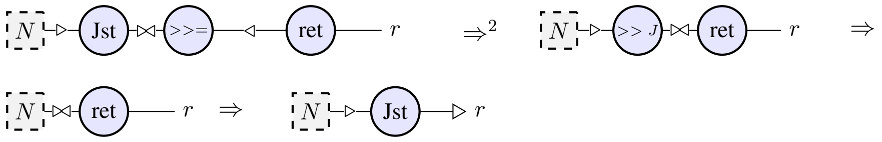
Figure 4.2: The Typing of the *Maybe* IN monad

4.2.2 Correctness of the Maybe Monad

We now show that the rules for the *Maybe* IN monad satisfy the equalities in Definition 4.1.1. The following reduction sequence shows equality (1), where N is an arbitrary net such that reduction is possible (i.e., there is an active pair connection between N and ret). We omit the variadic ports of α to simplify the graphical representation.



Equality (2) is proved by the following sequence:



4.3 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 4.3.

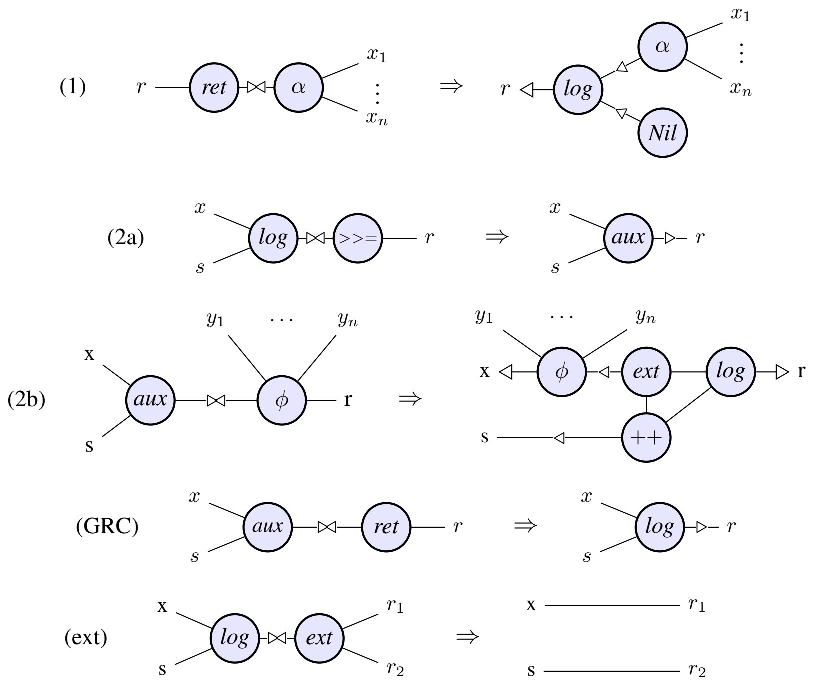


Figure 4.3: The rules of the *Writer* monad

Proposition 4.3.1. $\Rightarrow_{Writer_{DPC}}$ *satisfies the uniform confluence property.*

Proof. Same as Proposition 4.2.1 □

Port and Rule Types

The environments in Figure 4.4 type 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. The $Writer$ rules can be typed as follows. Again, we use subscripts to distinguish the type variables of both agents.

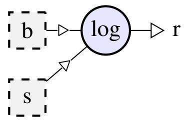
$$\begin{aligned}
\varepsilon(log) &= \{log(\tau,\rho)^+, \tau^-, list(\rho)^-\}\\
\varepsilon(ret) &= \{\tau^-, log(\tau,\rho)^+\}\\
\varepsilon(>>=) &= \{log(\tau,\rho)^-, \tau^+\}\\
\varepsilon(aux) &= \{\tau^+, \tau^-, list(\rho)^+\}\\
\varepsilon(ext) &= \{log(\tau,\rho)^-, \tau^+, list(\rho)^+\}\\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

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

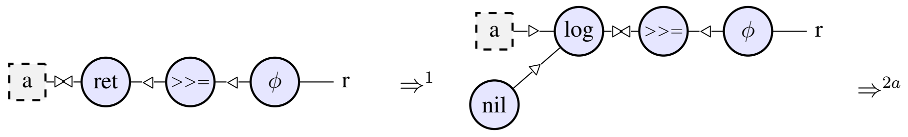
Figure 4.4: The Typing of the *Writer* monad

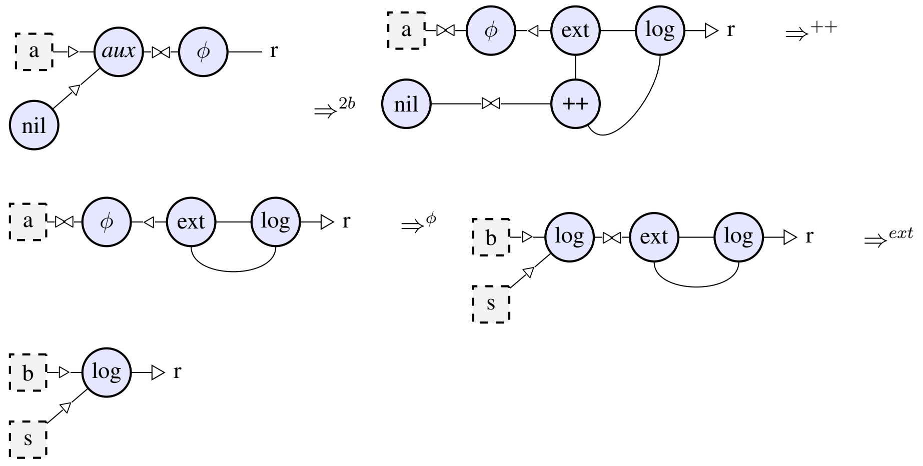
4.3.1 Correctness of the Writer Monad

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



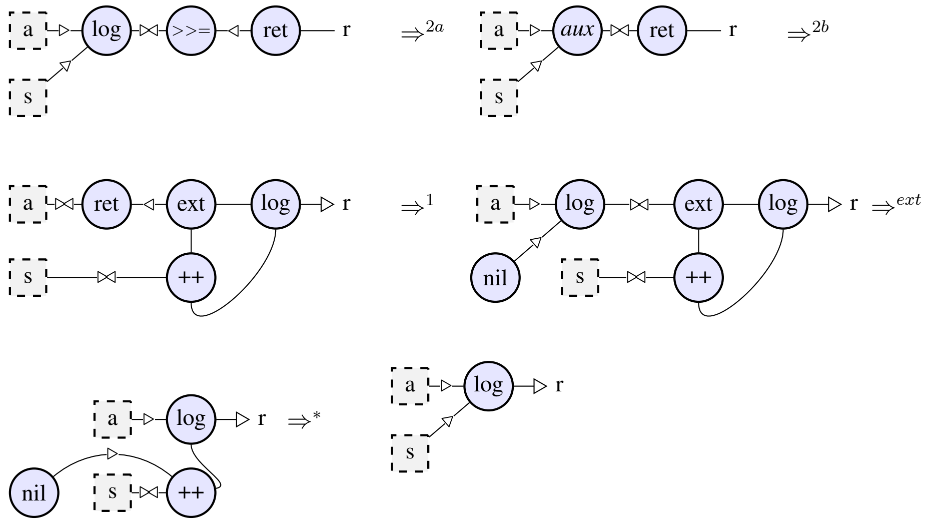
(the arbitrary nets a, b and s are represented by dashed squares). Again, we omit variadic ports for the sake of simplicity. The following reduction sequence shows equality (1):





The first step uses rule (1) or (2) depending on the concrete instance of a (0 or $S(x)$).

The following reduction sequence shows equality (2), which corresponds to *(m >>= return)* = *m* (where *m* = *(a,s)*):



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

As we use the same modeling approach as with the $Maybe$ monad, the third monad law holds due to Proposition 4.1.2.

4.4 The State Transformer Monad

The *State Transformer* monad is another well-known monad in Haskell that essentially generalizes the writer monad. 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 s a`, 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., [34,40]). 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 [1]. The *State Transformer* INS, including rules for explicit function application and handling of pairs, is shown in Figure 4.5.

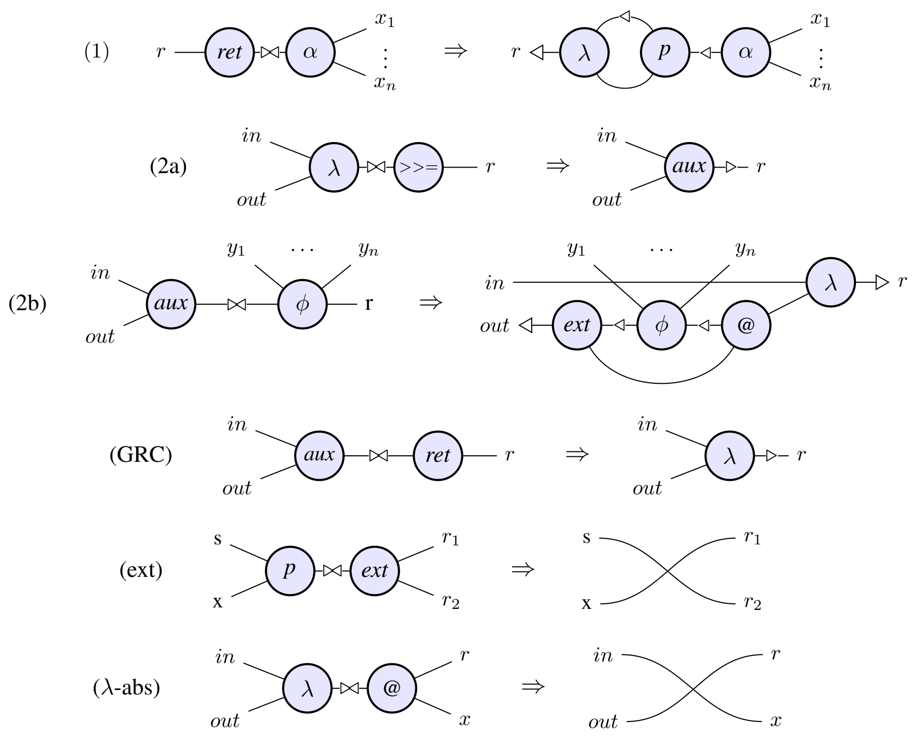


Figure 4.5: The State Transformer IN Monad

[1]We conjecture that the *rule archetypes* of [40] can be used to improve this - cf. Section 6

Like the INS for the *Maybe* monad, it is straightforward to show that the set of rules satisfies uniform confluence.

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. The corresponding rule types are shown on the right. Again, we use subscripts to distinguish the type variables of both agents.

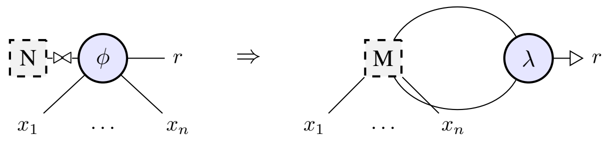
$$
\begin{aligned}
\varepsilon(\lambda) &= \{state(\tau,\rho)^+, \tau^+, pair(\rho,\tau)^-\} \\
\varepsilon(ret) &= \{\tau^-, state(\rho,\tau)^+\} \\
\varepsilon(>>=) &= \{state(\tau,\rho)^-, \tau^+\} \\
\varepsilon(aux) &= \{\tau^+, \tau^+, pair(\tau,\rho)^-\} \\
\varepsilon(ext) &= \{pair(\tau,\rho)^-,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

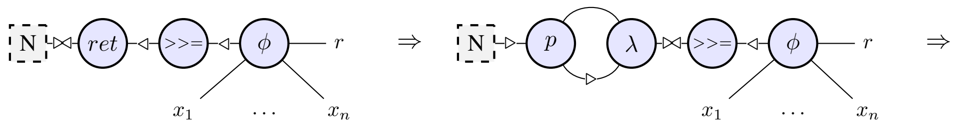
Figure 4.6: The Typing of the *State Transfomer* IN monad

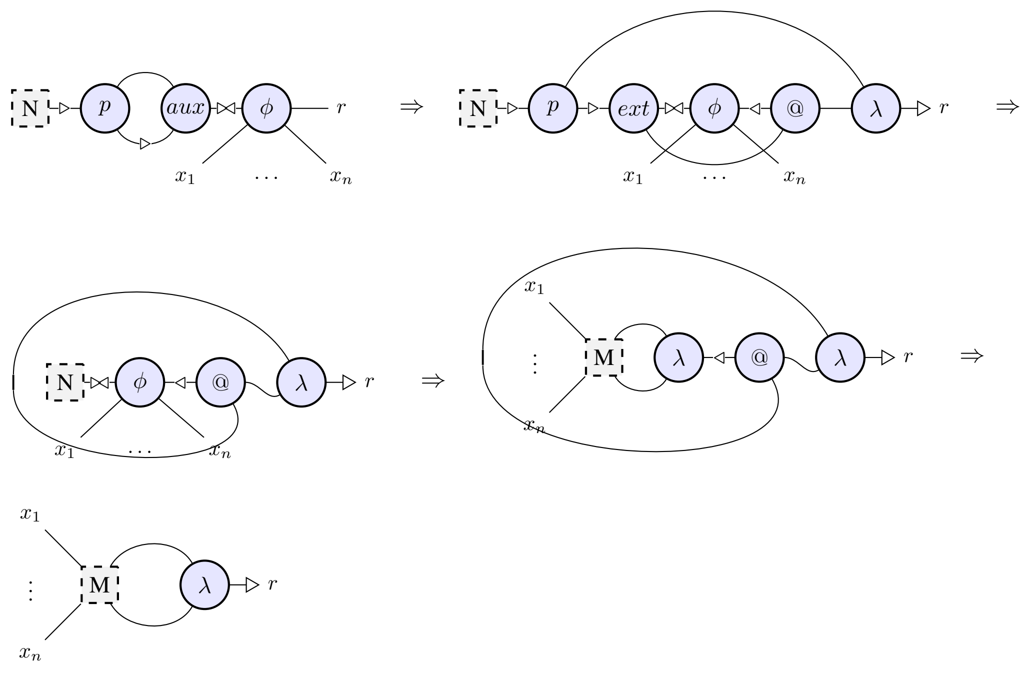
Due to the restriction on interaction rule LHSs, both monad rulesets feature auxiliary agents and rules. This can be improved by using rules with *nested patterns*: nested patterns are a conservative extension of interaction nets. They allow for more complex rule patterns while preserving uniform confluence. For more information, we refer to [17, 20].

4.4.1 Correctness of the State Transformer Monad

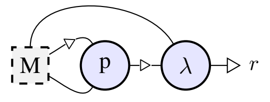
We now show that the monad laws hold for the *State Transformer* IN monad. For equality (1) of Definition 4.1.1, we assume `f a = \s -> (b,t)`, or in IN form:



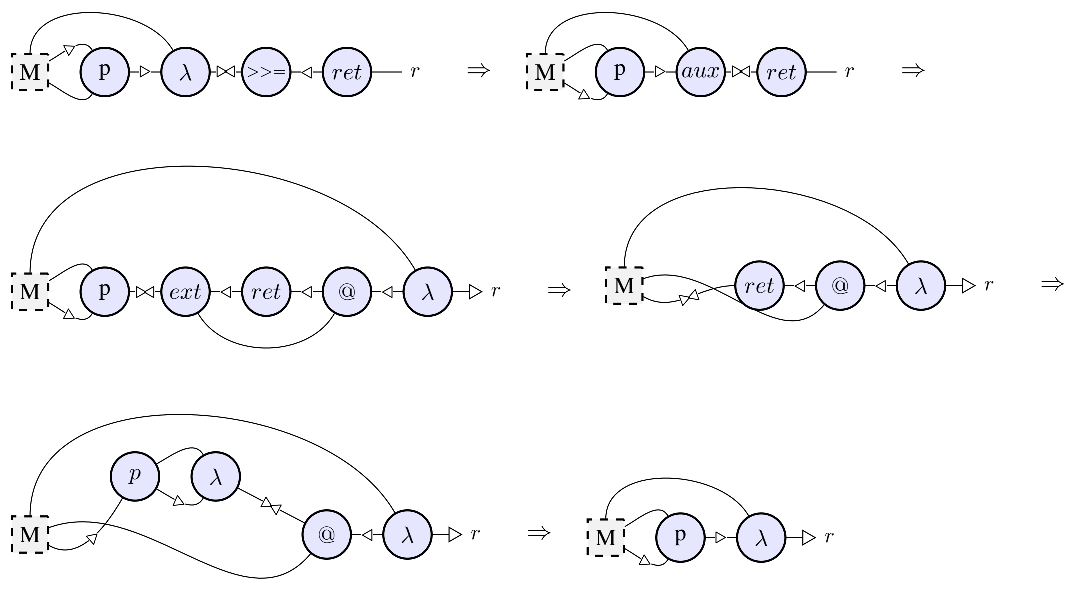




To show the second equality, we represent the monad object `m` by the following net:



We then show law 2 by the following reduction sequence:



4.5 The List Monad

The List monad is used to sequentialize functions that take a single value as input and return an ordered sequence of values as output. For example, consider two functions `f :: a -> [b]` and `g :: b -> [c]`, where `[t]` denotes a list of type `t`. As each function takes just a single argument but returns a list, the operators of the list monad operators are used to concatenate them. In essence, the function `g` is simply applied to all results of `f`, combining all results of `g`'s applications in a single list.

In Haskell, the List monad is defined as follows:

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

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

4.5.1 Adaptation to Interaction Nets using Nested Patterns

We use the same agents to represent lists as we did for the *Writer* monad. The rules for the *List* IN monad are shown in Figure 4.7. To give an example of GINP rules (see Section 3.7), we use nested patterns for the *List* monad. This results in the absence of auxiliary rules and a more concise representation. Interestingly, we do not need any GRC rules, as the generic nested patterns do not overlap.

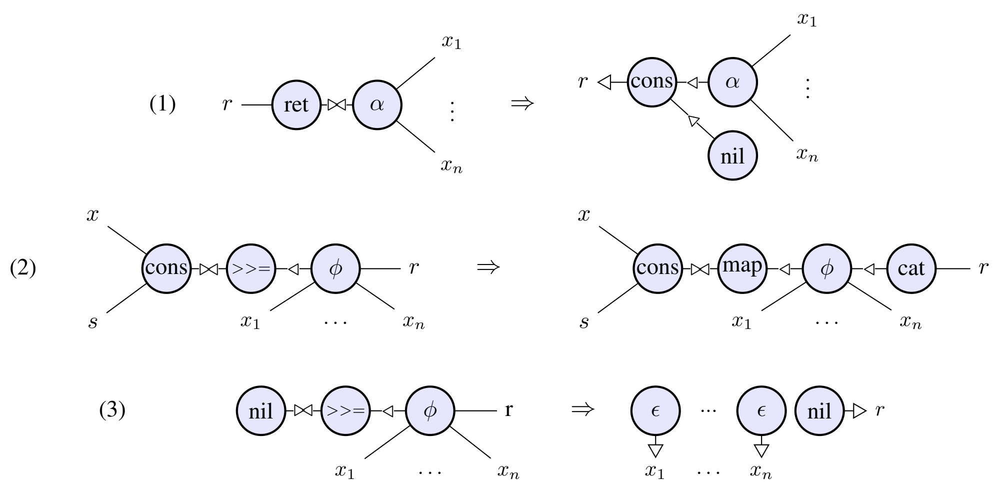
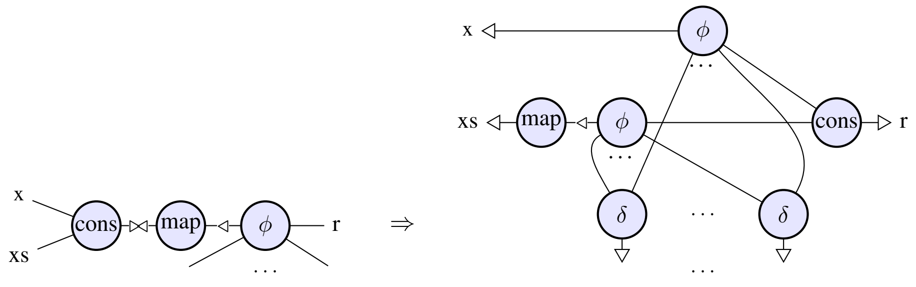


Figure 4.7: The List IN monad

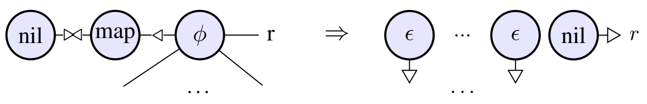
Note that the GINP rules of the *List* monad only have one generic agent at a leaf position of the nested pattern tree and can hence be translated to non-nested rules (see Proposition 3.7.5.1).

The rule for *return* is obviously similar to the corresponding rule of the *Writer* monad. Instead of a pair of a value and an empty list, a singleton list is returned. The interaction rule for $>>=$ behaves analogously to the function definition in Haskell. Rule (3) models the behavior of *bind* when connected to an empty list. In this case, the empty list is simply propagated and the agent α is discarded. Rule (2) uses the auxiliary agents *map* and *cat*. Their corresponding interaction rules are defined as follows:

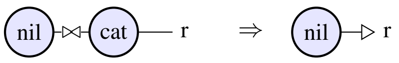
(map1)



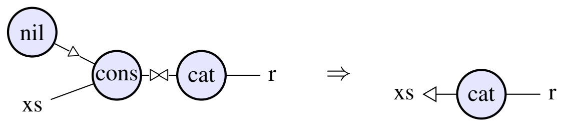
(map2)



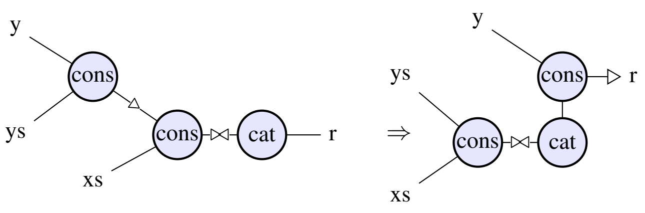
(cat1)



(cat2)



(cat3)



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

Proposition 4.5.1. *Let R be the set of interaction rules consisting of the $List$ IN monad, map and cat rules. The reduction relation induced by R is uniformly confluent.*

Proof. R is well-formed: First, no LHS is a subnet of another. Second, all LHSs can be added to a sequential set. Note that only two rules share the same active pair (the second and third rule of (cat). However, these LHSs can be added to a sequential set as the respective nested agent (nil and $cons$) is connected to the same port. This satisfies the sequential set condition. Hence, by Proposition 2.1.1.1, the reduction relation induced by R is uniformly confluent. □

4.5.2 Typing the List Monad

When typing the list monad, we encounter an issue with our previous definition of the *map* rules. Unfortunately, the rule *(map1)* cannot be well-typed (as in Definition 3.2.1.6): The interface port r has a different on the LHS than on the RHS. While r is type $list(\rho)^+$ on the RHS (the principal port of $cons$ is connected to it), it has ϕ's type ρ^+ on the LHS. The same issue occurs in *(map2)*. In contrast to *bind*, the functional argument of *map* does not have the same output type as the function itself. As a quick solution to this problem, we add an auxiliary agent *mAux* to the nested patterns of map such that it can be well-typed (see Figure 4.8).

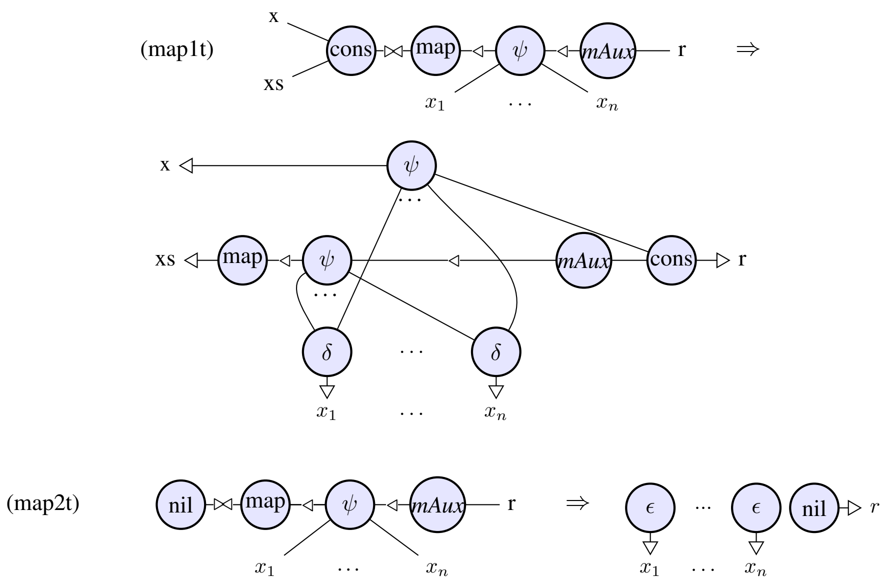


Figure 4.8: A well-typed version of *map*

The typing of the *List* features nested rule types as introduced in Section 3.7.

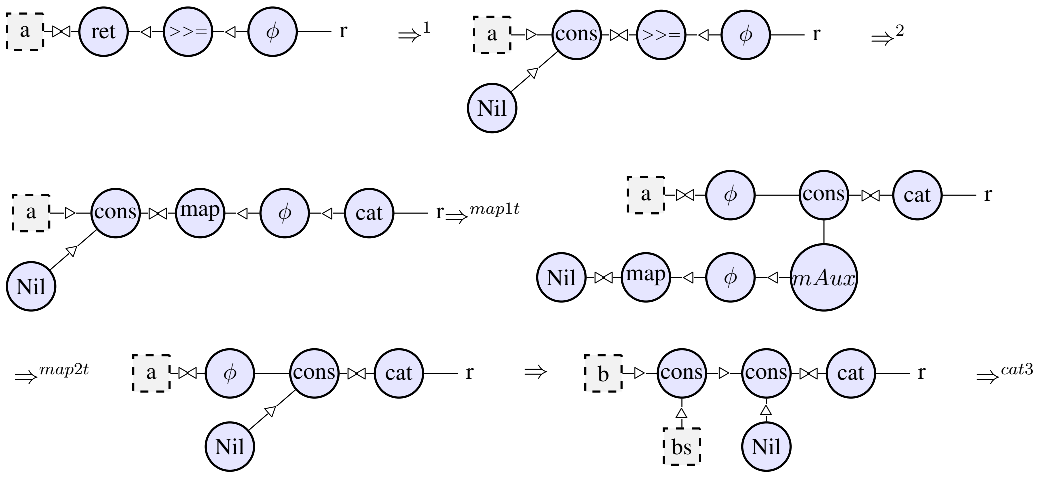
4.5.3 Correctness of the List Monad

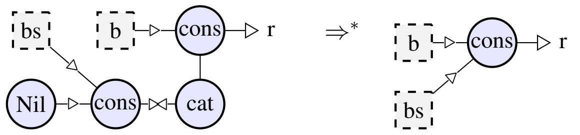
We show the correctness of the $List$ monad laws analogously to the previous examples. As for the Writer monad, we use arbitrary but fixed nets (denoted by dashed squares) in the reduction sequences showing the monad laws. Let a,b,bs be arbitrary nets such that *f a* = *Cons b bs*. The following reduction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

$$
\begin{aligned}
\varepsilon(cons) &= \{list(\tau)^+, \tau^-, list(\tau)^-\}\\
\varepsilon(nil) &= \{list(\tau)^+\}\\
\varepsilon(ret) &= \{\tau^-, list(\tau)^+\}\\
\varepsilon(>>=) &= \{list(\tau)^-, \tau^+\}\\
\varepsilon(map) &= \{list(\tau)^-, \tau^+\}\\
\varepsilon(mAux) &= \{\tau^-, list(\tau)^+\}\\
\varepsilon(cat) &= \{list(list(\tau))^-, list(\tau)^+\}\\
\varepsilon(\alpha) &= \{\tau^+, *\}\\
\varepsilon(\phi) &= \{\tau^-, list(\rho)^+, *\}\\
\varepsilon(\psi) &= \{\tau^-, \rho^+, *\}
\end{aligned}
$$

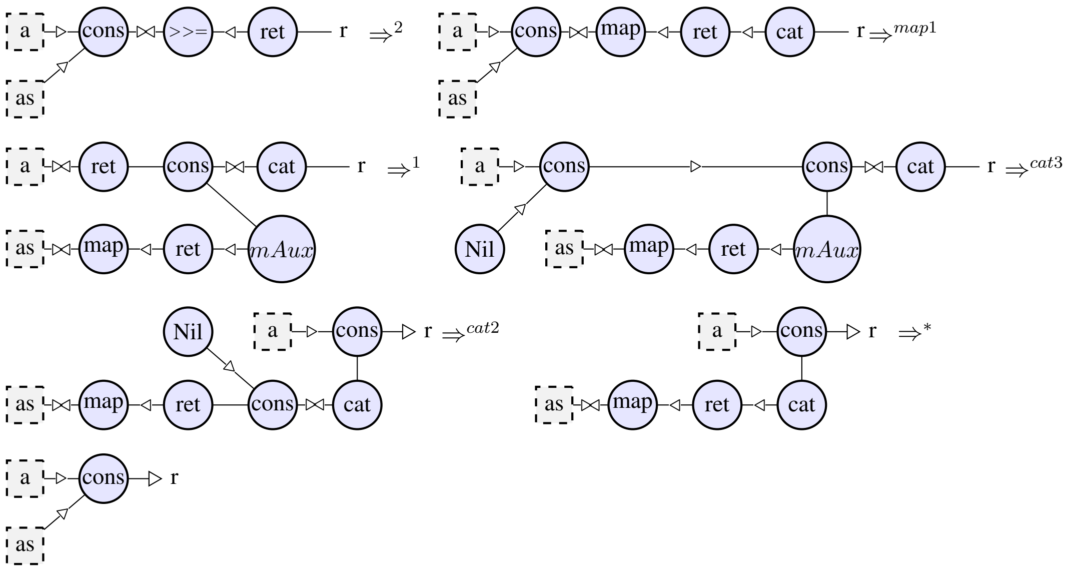
$$
\begin{aligned}
RT(1) &= \{\varepsilon(ret), \varepsilon(\alpha), \{\}\}\\
RT(2) &= \{\varepsilon(cons), \varepsilon(>>=), \varepsilon(\phi), \{\tau_1 \mapsto \tau_2, \tau_2 \mapsto \tau_3\}\}\\
RT(3) &= \{\varepsilon(nil), \varepsilon(>>=), \varepsilon(\phi), \{\tau_1 \mapsto \tau_2, \tau_2 \mapsto \tau_3\}\}\\
RT(map1t) &= \{\varepsilon(cons), \varepsilon(map), \varepsilon(\psi), \varepsilon(mAux), \{\tau_1 \mapsto \tau_2, \tau_2 \mapsto \tau_3, \rho_3 \mapsto \tau_4\}\}\\
RT(map2t) &= \{\varepsilon(nil), \varepsilon(map), \varepsilon(\psi), \varepsilon(mAux), \{\tau_1 \mapsto \tau_2, \tau_2 \mapsto \tau_3, \rho_3 \mapsto \tau_4\}\}\\
RT(cat1) &= \{\varepsilon(nil), \varepsilon(cat), \{\tau_1 \mapsto \tau_2, \}\}\\
RT(cat2) &= \{\varepsilon(cons), \varepsilon(cat), \varepsilon(nil), \{\tau_1 \mapsto \tau_2, \tau_1 \mapsto \tau_3\}\}\\
RT(cat3) &= \{\varepsilon(cons), \varepsilon(cat), \varepsilon(nil), \{\tau_1 \mapsto \tau_2, \tau_1 \mapsto \tau_3\}\}
\end{aligned}
$$

Figure 4.9: The Typing of the *List* IN monad





The last step is achieved by multiple applications of the (cat) rules (one for each element in *bs*). Equality (2 can be shown by the following reduction:



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

As with the previous monads, law 3 holds due to Proposition 4.1.2.

4.6 Combining the IN monads

Finally, we show that all IN monads can be added to a single ruleset without losing uniform confluence.

Proposition 4.6.1. *Let R be the set of interaction rules consisting of the $Maybe$, $Writer$ and $List$ monad including auxiliary agents. The reduction relation induced by R is uniformly confluent.*

Proof. R is well-formed: No LHS is a subnet of another. In addition, all LHSs can be added to a sequential set: As shown before, the rules of the $List$ monad can be added to a sequential set. The $Maybe$, *Writer* and *State Transformer* monad satisfy uniform confluence as they consist of ordinary rules only. Therefore, R satisfies the sequential set condition. While some of the rules of the monads have the same active pair (e.g. *ret* rules), there are no overlaps if we use typed nets and typed rule matching. By Proposition 3.6.4, the reduction relation induced R is uniformly confluent. □

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

4.7 Monads in Interaction Nets: Summary

We have shown how to define a notion of monad in interaction nets that is conceptually very close to the their usage in functional programming. The described IN monads handle typical side effects such as exception handling, state manipulation and logging. Our previously introduced extensions generic rules, nested patterns and the updated type system have been employed. The careful reader might wonder why we have given no example of the most prominent side effect, namely *input/output*. The *I/O* monad can be seen as a special case of the state transformer monad. Indeed, Haskell's `IO` monad is implemented by sequentially manipulating an object representing the *state of the world* (also compare Section 2.6.2). A similar *I/O* IN monad can be defined in a straightforward way.

CHAPTER 5

Implementation and Evaluation

In this chapter, we describe our efforts to implement our theoretical results in prototype programming languages. We introduce two systems, *inets* [23] and *ingpu* [16]. The first part of the chapter deals with *inets* with a focus on the implementation of nested patterns and generic rules. The sections on nested patterns are based on joint work with Abubakar Hassan and Shinya Sato [17].

The second part of the chapter discusses *ingpu*, an experimental GPU-based implementation of INs with a focus on maximally parallel evaluation.

5.1 The *inets* Language

The *inets* system 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 are implemented by the following piece of code (We use the syntax of the *inets* language, where equations are denoted by >< on the LHS (i.e., the active pair), and by $\sim$ on the RHS):

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

We represent nets as a comma separated list of agents pairs. Obviously, this is based on the textual definition of interaction nets in the lightweight calculus. For example, recall the rules for computing the last element of a list in Figure 3.2. The LHS net of the first rule can be expressed as follows:

```
  p~Lst(r),p~Cons(x,xs),x~1,xs~Nil
```

The symbol '$\sim$' denotes the principal port of the agent ('$=$' in the lightweight calculus). The variables `p`, `x` and `xs` are used to model the connection between two ports. As in the lightweight calculus, all variable names occur at most twice: this limitation corresponds to the requirement that it is not possible to have two edges connected to the same port in the graphical setting. If a name occurs once, then it corresponds to the free ports of the net (r is free in the above net). If a name occurs twice, then it represents an edge between two ports. In this latter case, we say that a variable is *bound*.

The syntax above can be simplified by replacing equals for equals:

```
Lst(r)∼Cons(1,Nil)
```

In this notation the general form of an active pair is $\alpha(\ldots) \sim \beta(\ldots)$.

We represent rules by writing $l \Longrightarrow r$, where l is the net on the left of the rule, and r is the resulting net. In particular, we note that l will always consist of two agents connected at their principal ports. As an example, the rules in Figure 3.1 are written as:

```
Lst(r) >< Cons(x,xs) ⟹ xs∼Aux(x,r)
Aux(x,r) >< Nil ⟹ x∼r
Aux(x,r) >< Cons(y,ys) ⟹ Lst(r)∼Cons(y,ys), ϵ ∼x
```

The names of the bound variables in the two nets must be disjoint, and the free variables must coincide, which corresponds to the condition that the free variables must be preserved under reduction.

In order to represent INP rules, we allow the LHS of a rule to contain more than two agents. As an example, the set of INP rules in Figure 3.2 can be written as:

```
Lst(r) >< Cons(x,xs), xs∼Nil ⟹ r∼x
Lst(r) >< Cons(x,xs),xs∼Cons(y,ys) ⟹ x∼ ϵ,p∼Lst(r),p∼Cons(y,ys)
```

or in a simplified form:

```
Lst(r) >< Cons(x,Nil)⟹ r∼x
Lst(r) >< Cons(x,Cons(y,ys))⟹ x∼ ϵ,Lst(r)∼Cons(y,ys)
```

Other language constructs The *inets* system provides other language constructs, including modules, built-in operations on agent values and a basic version of I/O (with side effects). These constructs remain unaffected by our extension and will hence not be discussed here. For a detailed description of the additional language features, we refer to [18, 39].

5.2 Nested Pattern Translation

The runtime of *inets* handles ordinary interaction (ORN) rules only. Therefore, all INP rules need to be translated into sets of ORN rules during compilation. In this section, we give an overview of this translation. In general, the *inets* compiler reads programs in our source language and builds the corresponding abstract syntax tree (AST). On the basis of the AST, the compiler generates byte code which can be executed by an abstract machine or be further compiled into C source code.

Our translation function rewrites subtrees of the AST that represent INP rules into sets of subtrees that represent ordinary interaction rules. Overall, our translation function is similar to the compilation schemes defined in the original paper [20]. The main objective of the algorithm is to check the set of rules for the well-formedness property of INP rules (Definition 3.6.3). We summarise the translation algorithm in the following steps:

1. A rule is found in the AST. This rule can be either ORN or INP. All other nodes of the AST that are not rules (imports, variable declarations,. . .) are ignored.
2. Check if the LHS is not a subnet of a previously translated LHS (and vice versa if the rule is INP). This is the first part of verifying the well-formedness property. We discuss this verification in Section 5.3.1.
3. If the rule is not INP, return.
4. If the rule is INP: check if the current and all previous nested active pairs can be added to a sequential set. This is the second part of verifying the well-formedness property (Section 5.3.2).
5. If both checks are passed, translate the rule (else, exit with an error message):
    a) Resolve the first nested agent of the rule's active pair.
    b) Add an auxiliary rule to the AST.
    c) The remaining nested agents are not (yet) translated. They are resolved by translating the auxiliary rule.

   We describe the translation algorithm in Section 5.4.
6. traverse the AST until the next (unprocessed) rule is found.

This algorithm allows for an arbitrary number of nested patterns (i.e., the number of nested agents in the LHS of an INP rule) and an arbitrary pattern depth.

5.3 Verifying the Well-Formedness Property

Our verification algorithm (see below) consists of two parts which correspond to the two constraints of the well-formedness property. We verify that the set of nested active pairs in a given *inets* program are both disjoint and sequential.

We need a precise notion of the position of an agent in a nested active pair in order to show the well-formedness property. This will be done by computing lists of positions of each symbol in a net. We use the notation `[]` for the empty list, $[1, \dots, n]$ for a list of n elements and $ps_1 @ ps_2$ to append two lists.

Definition 5.3.0.1 (**Position Set**). *Let* $l \implies r$ *be a rule in* inets. *We define the function* $\mathrm{PosSym}(l)$ *that, given a nested active pair, will return a set of pairs* (ps, u) *where* ps *is a* *list that represents the* position of a symbol u *in* l.

$$\begin{array}{lcl}
\mathrm{PosSym}(\alpha\,\texttt{(}t_1\texttt{,}\ \dots\texttt{,}t_n\texttt{)}\ \texttt{><}\ \beta\,\texttt{(}s_1\texttt{,}\ \dots\texttt{,}s_m\texttt{)}\texttt{)} & = & \mathrm{PosSym}_t(\texttt{[}1,1\texttt{]},\ t_1) \\
 & & \cup \dots \cup \mathrm{PosSym}_t(\texttt{[}1,n\texttt{]},\ t_n) \\
 & & \cup\ \mathrm{PosSym}_t(\texttt{[}2,1\texttt{]},\ s_1) \\
 & & \cup \dots \cup\ \mathrm{PosSym}_t(\texttt{[}2,m\texttt{]},\ s_m) \\
\mathrm{PosSym}_t(ps,\ x) & = & \emptyset \\
\mathrm{PosSym}_t(ps,\ \alpha\,\texttt{(}\bar{x}\texttt{)}) & = & \{(ps,\alpha)\} \\
\mathrm{PosSym}_t(ps,\ \alpha\,\texttt{(}t_1\texttt{,}\ \dots\texttt{,}t_n\texttt{)}) & = & \mathrm{PosSym}_t(ps\ \texttt{@}\ \texttt{[}1\texttt{]},\ t_1) \\
 & & \cup \dots \cup\ \mathrm{PosSym}_t(ps\ \texttt{@}\ \texttt{[}n\texttt{]},\ t_n)
\end{array}$$

where the sequence of terms $t_1, \dots, t_n$ *in* $\mathrm{PosSym}_t(ps,\ \alpha\,\texttt{(}t_1\texttt{,}\ \dots\texttt{,}t_n\texttt{)})$ *contain at least one term which is not a variable; and* $\bar{x}$ *is a sequence of zero or more variables.*
The function $\mathrm{Pos}(l)$ *returns a set of lists that represent the position of each nested agent in a nested active pair:*

$$\begin{array}{lcl}
\mathrm{Pos}(l) & = & \pi_1(\mathrm{PosSym}(l)) \\
\pi_1(\emptyset) & = & \emptyset, \\
\pi_1(\{(ps,s)\} \cup A) & = & \{ps\} \cup \pi_1(A - \{(ps,s)\}).
\end{array}$$

We extend these operations into the sequence $l_1, \dots, l_k$ *of LHS of rules as follows:*

$$\begin{array}{lcl}
\mathrm{PosSym}(l_1,\dots,l_k) & = & \mathrm{PosSym}(l_1) \cup \dots \cup \mathrm{PosSym}(l_k), \\
\mathrm{Pos}(l_1,\dots,l_k) & = & \mathrm{Pos}(l_1) \cup \dots \cup \mathrm{Pos}(l_k)
\end{array}$$

Example 5.3.0.2. *For each rule in Figure 3.2, the sets of positions of nested agent pairs are as follows:*

- $\mathrm{PosSym}(\texttt{Lst(r) >< Cons(x,Nil)})$

  $= \mathrm{PosSym}_t(\texttt{[}1,1\texttt{]},\texttt{r}) \cup \mathrm{PosSym}_t(\texttt{[}2,1\texttt{]},\texttt{x}) \cup \mathrm{PosSym}_t(\texttt{[}2,2\texttt{]},\texttt{Nil})$
  $= \{(\texttt{[}2,2\texttt{]},\texttt{Nil})\}$

- $\mathrm{Pos}(\texttt{Lst(r) >< Cons(x,Nil)}) = \{\texttt{[}2,2\texttt{]}\}$

- $\mathrm{PosSym}(\texttt{Lst(r) >< Cons(x,Cons(y,ys))})$

  $= \mathrm{PosSym}_t(\texttt{[}1,1\texttt{]},\texttt{r}) \cup \mathrm{PosSym}_t(\texttt{[}2,1\texttt{]},\texttt{y}) \cup \mathrm{PosSym}_t(\texttt{[}2,2\texttt{]},\texttt{Cons(y,ys)})$
  $= \{(\texttt{[}2,2\texttt{]},\texttt{Cons})\}$

- $\mathrm{Pos}(\texttt{Lst(r) >< Cons(x,Cons(y,ys))}) = \{\texttt{[}2,2\texttt{]}\}$

5.3.1 Verifying the Subnet Property

Verifying the subnet property is straightforward. Since rules are represented as trees (subtrees of the AST), it is easy to verify if one rule's LHS is a subtree of another. We compute the LHS subtree relation of the current rule P (to be translated) against all the rules Q which have already been translated. If P is in ORN, we verify the subtree relation in one direction only: the LHS of an INP rule cannot be a subnet of the LHS of an ORN rule. Otherwise we verify the subtree relation in both directions: P against Q and Q against P. The case of two ORN rules with the same active agents is handled by the compiler at an earlier stage. If the current rule's LHS is not a subnet of any previous rules' LHS's, we add it to the set of previous rules.

Note that we consider a tree to be a subtree of another tree up to alpha conversion, i.e., variable names are not considered.

5.3.2 Verifying the Sequential Set Property

The check for the sequential set property is a bit more complicated than the subnet one. According to the definition of the well-formedness property, there must exist a sequential set that contains all nested active pairs in a given set of INP rules. Rather than attempting to construct such a sequential set, the algorithm tries to falsify this condition: it searches (exhaustively) for two nested patterns that cannot be in the same sequential set. This is done as follows:

For the current nested pattern P and all previously verified patterns Q with the same active agents:

1. Compare the sets $\mathrm{Pos}(P)$ and $\mathrm{Pos}(Q)$. We only consider the positions of agents at this point, not the agents themselves.
2. If one set is a subset of another, P and Q can be added to a sequential set [1]. P is added to the set of previous nested patterns.
3. Else, we compare the actual nested agents at the common positions $CP = \mathrm{Pos}(P) \cap \mathrm{Pos}(Q)$.
4. If for each element $p \in CP$, α and β are the same where $(p, \alpha) \in \mathrm{PosSym}(P)$ and $(p, \beta) \in \mathrm{PosSym}(Q)$, no sequential set can contain P and Q, as $P \equiv \langle M, x \sim \alpha(\ldots), \mathbf{a}\rangle$, $Q \equiv \langle M, y \sim \beta(\ldots), \mathbf{b}\rangle$ with $x \neq y$.
5. Else, P and Q can be added to a sequential set. P is added to the set of previously verified nested patterns.

It is straightforward to see that after the full traversal of the AST, all possible pairs of nested patterns are considered. Hence, the search for a pair that violates the sequential set property is exhaustive.

Proposition 5.3.2.1. *Let R be a set of INP rules. R is well-formed $\Leftrightarrow$ R is correctly verified to be well-formed using our verification algorithm.*

Proof. $\Leftarrow$:

Assume R is not well-formed and passes the well-formedness checks. We proceed by a complete case distinction (according to the definition of well-formedness):

[1] Note that P and Q have already passed the subnet check at this point. This means that (some of) the nested agents at the common positions are different. Hence, P and Q cannot give rise to a critical pair.

Case 1. There exist two rules $P \implies N, Q \implies M \in R$ where P is a subnet of Q. But then, the pair (P, Q) is tested for the subnet relation (Section 5.3.1). Hence, R does not pass the well-formedness check.

Case 2. There exist two rules $A \implies N, B \implies M \in R$ where $A \equiv \langle P, x \sim \alpha(\ldots)\rangle, B \equiv \langle P, y \sim \beta(\ldots)\rangle$ for $x \neq y$. But since all pairs of nested patterns are checked for the sequential set property (Section 5.3.2), (A, B) will be detected. Hence, R does not pass the check.

In both cases, we reach a contradiction to the assumption above, hence it cannot be true.

$\Rightarrow$:

Assume R does not pass the well-formedness check, but is well-formed. Again, there are only two cases:

Case 1. R does not pass the check because $\exists P \implies N, Q \implies N' \in R$ where P is a subnet of Q. But then, R is not well-formed (by the definition of well-formedness).

Case 2. R does not pass because are two rules $A \implies N, B \implies M \in R$ where $A \equiv \langle P, x \sim \alpha(\ldots)\rangle, B \equiv \langle P, y \sim \beta(\ldots)\rangle$ for $x \neq y$. Then, there is no sequential set that contains both A and B. Hence, R is not well-formed.

Again, we reach a contradiction to the assumption above in either case.

$\square$

5.4 Nested Rule Translation

We now describe the translation algorithm from nested to ordinary rules in more detail. As mentioned earlier, we translate INP rules to ORN rules by rewriting the AST. We perform a pre-order traversal of the AST and identify nodes that represent INP rules. Once we find an INP rule, we replace its nested agents with a fresh (variable) node n and replace the subtree that represents the RHS of the rule with a new tree N_t. The nested agents and the RHS of the rule are stored for later processing. The tree N_t represents an active pair between n and a newly created auxiliary agent Aux. This auxiliary agent holds all the agents and attributes of the original active pair, with the exception of the former variable agent.

Now, we create an auxiliary rule with an active pair between Aux and the current nested agent (initially connected to the interacting agent). We set the RHS of the auxiliary rule to be the RHS of the original INP rule. Finally, we add this auxiliary rule to the system.

Note that the auxiliary agent in the new rule may still contain additional nested agents, i.e., the auxiliary rule may be INP. Hence, the translation algorithm recursively translates the generated rules until the LHS of each of the generated rules contains exactly two agents. The idea behind this is to resolve one nested agent per translation pass. Further nested agents are processed when the translation function reaches the respective auxiliary rule(s).

We can formalise the translation algorithm as a function $translate(\mathcal{R}, U, S)$, where $\mathcal{R}$ denotes the input set of interaction rules and U and S are *stores*. Intuitively, the components U and

S are used to store previously processed rule patterns in order to verify the subnet and sequential set properties respectively. The function $translate$ is defined in Figure 5.1 (in pseudo-code notation), where `FAIL` denotes termination of the program due to non well-formedness of $\mathcal{R}$:

```
translate([],$U$,$S$) = []
translate((P $\Longrightarrow$ N):$\mathcal{R}$,$U$,$S$) =
if (P is a subnet of any Q $\in U$ or vice versa)
    FAIL
else if (P is ORN)
    (P$\Longrightarrow$N):translate($\mathcal{R}$,P:$U$,$S$)
else
    if (P cannot be added to a sequential set with
any Q $\in S$)
        FAIL
    else
    (P'$\Longrightarrow$(PX$\sim$p)):translate((PX >< A $\Longrightarrow$ N):$\mathcal{R}$,P:$U$,P:$S$)
        where
            p = position of the first nested agent of P
            A = the nested agents at position p
            P' = P with all nested agents replaced by variable
                 ports
            PX = auxiliary agent that contains all ports of P
                 except p
```

Figure 5.1: The INP to ORN translation algorithm

Example 5.4.0.1. *Consider the interaction rules from Figure 3.2. $\mathcal{R}$ consists of two rules:*

1. `Lst(r) >< Cons(x,Nil)` $\Longrightarrow$ `r`$\sim$`x`
2. `Lst(r) >< Cons(x,Cons(y,ys))` $\Longrightarrow$ `x`$\sim\epsilon$`,Lst(r)`$\sim$`Cons(y,ys)`

The translation works as follows:

- *Rule 1 is INP (due to the `Nil` agent).*
  Its LHS is checked for the subnet property. As there are no previous rules in U, the check is passed.
- *As the rule is INP, it is checked for the sequential set property. Again, there are no previous rules in S, hence it passes the check.*
- *The rule is transformed, introducing the auxiliary agent `Lst_Cons`*

  1. `Lst(r) >< Cons(x,var0)` $\Longrightarrow$ `Lst_Cons(r,x)`$\sim$`var0`

- *A new auxiliary rule is added to $\mathcal{R}$:*

```
3. Lst_Cons(r,x) >< Nil ⟹ r∼x
```

The LHS pattern of rule 1 is added to U and S.

- *Rule 2 is INP (due to the nested* `Cons` *agents).*
  Its LHS is checked for the subnet property. The only LHS pattern in U is `Lst(r) >< Cons(x,Nil)`*. Due to different agents at the second auxiliary port of* `Cons`*, the LHSes cannot be subnets of one another. The check is passed.*

- *Rule 2 is checked for the sequential set property. First, the positions of nested agents of Rule 1 and 2 are compared. Since they are the same (both have their nested agent at the second auxiliary port of* `Cons`*), they can be added to a sequential set. There are no further rules in S, hence the check is passed.*

- *The rule is transformed and another auxiliary rule is added to $\mathcal{R}$:*

```
2. Lst(r) >< Cons(x,var1) ⟹ Lst_Cons(r,x)∼var1
4. Lst_Cons(r,x) >< Cons(y,ys) ⟹ x∼ ϵ,Lst(r)∼Cons(y,ys)
```

- *Rule 3 is ORN.*
  Its LHS is checked for the subnet property. As there are no with the same active agents in U, the check is passed. The LHS pattern of Rule 3 is added to U.

- *Rule 4 is ORN.*
  Its LHS is checked for the subnet property. Again, there are no rules with the same active agents in S. The check is passed and the LHS pattern is added to U.

This yields the translated set of rules

```
1. Lst(r) >< Cons(x,var0) ⟹ Lst_Cons(r,x)∼var0
2. Lst(r) >< Cons(x,var1) ⟹ Lst_Cons(r,x)∼var1
3. Lst_Cons(r,x) >< Nil ⟹ r∼x
4. Lst_Cons(r,x) >< Cons(y,ys) ⟹ x∼ ϵ,Lst(r)∼Cons(y,ys)
```

Note that the rules 1 and 2 are identical (save variable names). Since only one of them is needed, rule 2 is discarded. As expected, we get the set of ORN rules given in Figure 3.1.

Proposition 5.4.0.2. ***(Termination)*** *For a finite $\mathcal{R}$, $translate$ terminates.*

Proof. Let n be the number of rules in $\mathcal{R}$ and p be the sum of all nested agents of these rules. By a complete case distinction, we show that with each recursive call, $(n + p)$ decreases:

Case 1 The current rule is ORN. At the recursive call, it is removed from $\mathcal{R}$, hence n decreases by 1.

Case 2 The current rule has i nested agents ($i > 0$). The rule is removed from $\mathcal{R}$ and an auxiliary rule with exactly $i - 1$ nested agents is added to $\mathcal{R}$. Hence, with the recursive call p decreases by 1.

$translate$ terminates if $n = 0$. n only decreases if we encounter an ORN rule. Yet, since the number of nested agents for each rule is finite, all rules in $\mathcal{R}$ will be ORN after finitely many decreases of p. $\square$

5.4.1 Time and Space Complexity of $translate$

Time complexity To determine the time complexity of $translate$, we analyze the three main parts of the algorithm: the check for the subnet property, the check for the sequential set property and the actual rule translation. As a measurement of the input size, we consider the sum $n = r + a$, where r is the number of rules and a is the number of nested agents in the input set $\mathcal{R}$. The idea behind this notion of size is that $translate$ is invoked once for every interaction rule in the input. Additionally, each elimination of a nested agent yields a new auxiliary rule to be translated. Hence, $translate$ is called $n = r + a$ times.

The subnet property check compares all pairs of rule patterns. For an input set of size n, $\frac{n(n-1)}{2}$ checks are performed. Therefore, the verification of the subnet property is quadratic to the input size, or $O(n^2)$.

At first glance, the check for the sequential set property is similar to the one for the subnet property: all pairs of rule patterns need to be compared, resulting in a (worst-case) complexity of $O(n^2)$. However, the verification algorithm only considers pairs of patterns with the same active agents (see Section 5.3.2), which is usually only a fraction of all pairs of rule patterns. On average, this results in a less than quadratic complexity for the verification of the sequential set property.

The actual translation is executed once for each nested agent in the input set. The elimination of a single nested agent is not influenced by the total number of nested agents or rules. The rule translation is thus linear to the number of nested agents r.

This results in an overall time complexity of $O(n^2)$ for $translate$, which is mostly determined by the verification of the subnet property. Extended profiling on the implementation of the algorithm (including input sets with up to several thousand nested agents and rules) shows that also on practical cases the performance is bounded by a quadratic curve (with small constants involved).

Space complexity To analyze the space complexity of $translate$, we consider the population of the stores U and S. Every rule pattern of the input set is stored in U, whereas only nested patterns are stored in S. At the worst case, U and S will contain $2n$ rule patterns. This implies that space complexity is linear to the number of rules in the input. Again, this result is reflected by several profiling tests using practical examples.

5.4.2 Additional Language Features

The *inets* language offers some features that are not considered in the original definition of the nested pattern translation function. Some important examples are data values of agents (integers, floats, strings,...) realized via *external datatypes* (see Section 2.3)s, side effects (declaration and manipulation of variables, I/O) and conditions. These features are not involved in the process of nested pattern matching. Therefore, they do not need to be processed or changed by the translation function:

- with regard to nested pattern matching, data values can be considered as variable ports (they do not contain nested agents). Hence, they are unaffected by the translation.
- conditionals and side effects only occur in the RHS of a rule. Since the original RHS of an INP rule is propagated to the final auxiliary rule without a change, these features are not affected either.
- all auxiliary rules but the last one are responsible for pattern matching only, they do not do the "actual work" of the original rule. All special language features are simply passed to the next auxiliary rule.

5.4.3 The Implementation

The current version of inets can be obtained from the project's web page [23]. We have thoroughly tested the prototype implementation and developed several example modules. These examples include rule systems with a large number and depth of nested patterns as well as heavy use of state, conditionals and I/O. Additionally, we have designed several non well-formed systems in order to improve error handling.

5.5 Nested Patterns in *inets*: Summary

We have presented an implementation for nested pattern matching of interaction rules. The implementation closely follows the definition of nested patterns and their translation to ordinary rules. We have shown correctness, termination and complexity of the algorithm.

The resulting system allows programs to be expressed in a more convenient way rather than introducing auxiliary agents and rules to pattern match nested agents. We see this as a positive step for further extensions to interaction nets: future implementations of high-level language constructs can be built upon these more expressive rules.

5.6 Implementation of Generic Rules

In this section, we discuss the implementation of generic rules in *inets*. We can show that our implementation satisfies the generic rule constraints defined in Section 3.4.3. We will focus on the implementation of fixed generic rules and discuss which additions enable variadic rules.

The implementation of generic rules allows us to define interaction net systems similar to the Maybe monad in Example 3.4.5.1 (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;
```

In contrast to the mentioned example, this code only shows fixed generic agents instead of variadic agents. We will later discuss how the notion of arity unfolding (see Definition 3.1.2.2) can help us to implement variadic rules as a set of fixed generic rules.

5.6.1 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[$\phi_n$][agent2]
  if (rule == 0) {
    rule = ruleTable[$\phi_m$][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 $\phi_m$ be the generic agent of r's LHS
    n = arity(A)
    if (a generic rule with LHS B >< $\phi_n$ 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 3.4.3. 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 5.6.1.1. *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. As an example of a variadic rule in inets, consider the δ-rule:

```
Delta(d1,d2) >< ANY(Agents r) =>
   d1 ~ ANY(Agents s), d2 ~ ANY(Agents t),
   'r ~ Delta('s, 't);
```

The keyword `Agents` denotes a variadic range, whereas `'r` denotes the variadic name corresponding to the variadic range `r`.

5.7 Parallel Evaluation of Interaction Nets in CUDA/Thrust

In this section, we discuss the ongoing implementation of our GPU-based interaction nets evaluator *ingpu*. First, we give a quick introduction to CUDA/*Thrust* and motivate a GPU-based approach. We then describe the main components of the evaluator, the interaction and the communication phase. We discuss the performance drawbacks of our initial approach and describe measures to improve its efficiency.

5.7.1 CUDA and Thrust

The tool *ingpu* is written in C++ and CUDA. The latter is a language for GPU-based, general-purpose computation on NVIDIA graphics cards. The general flow of a program using the GPU for data-parallel computation is as follows: an array of input data sets is copied from the main memory (known as *host* memory) to the memory of the GPU (also referred to as *device*). A function (the *kernel*) is executed on the GPU in parallel on each individual data set. Finally, the array of results is copied back to main memory.

In general, implementing an algorithm on a GPU efficiently requires a considerable amount of low-level decisions: factors such as size of data structures, number of threads and thread block size can greatly influence performance. Fortunately, version 4.0 of CUDA introduced the *Thrust* library [21], which features high-level constructs for efficiently performing parallel computations. For example, *Thrust* provides the `transform()` function, which is similar to `map` in functional programming:

```
// allocate three device_vectors with 10 elements
thrust::device_vector<int> X(10), Y(10);
// compute Y = -X
thrust::transform(X.begin(), X.end(), Y.begin(),
   thrust::negate<int>());
```

Thrust is obviously inspired by the C++ STL: `device_vector` is a generic, resizable container residing in GPU memory. The arguments of `transform` are an input vector `X`, an output vector `Y` and a function object, here a built-in function that negates integers.

Other functions supplied by *Thrust* include parallel sorting, filtering and reduction. Reductions compute a single value based on an input list, e.g., the sum of its elements:

```
//sum the vector X, with initial fold value 0
s = thrust::reduce(X.begin(), X.end(), 0, thrust::plus<int>());
```

These functions are a convenient way to write parallel programs without the need for low-level tweaking. Our interaction nets evaluator *ingpu* is completely based on the *Thrust* library.

5.7.2 Motivation and Challenges

Why does a GPU-based implementation of interaction nets make sense in the first place? Several reasons can be given: first, the SIMD (Single Instruction, Multiple Data) model of GPUs is similar to the idea behind interaction nets. Several independent data sets are processed in parallel using the same instructions/program. This is analogous to reducing several active pairs with a common set of interaction rules. Second, the reduction of a single active pair is a fairly small computation, consisting only of a few lines of code. GPUs are optimized for running thousands of threads executing such small programs. Additionally, the number of active pairs existing at the same time may vary greatly through the execution of a program. An interaction nets evaluator should be able to dynamically and transparently scale this potential parallelism to the manycore hardware. Again, GPUs are a promising platform to achieve this.

However, the implementation of interaction nets on a GPU is a non-trivial task. In particular, it poses the following challenges:

Maintaining the net structure While active pairs can be reduced in parallel, they are not completely independent: they are connected in a net, and resolving these connections (via $\xrightarrow{com}$) is needed to generate further active pairs. Unfortunately, the choice of data structures in GPU memory is very limited (essentially just arrays). Moreover, typical GPU programs are most efficient for algorithms with regular data access (for example, dense matrix multiplication). This means that it is difficult to efficiently represent the irregular graph structure of an interaction net.

Varying output size of a reduction In general, the RHS of an interaction rule may be an arbitrarily large net. This implies that the result of a reduction of an active pair may vary in size, depending on the rule being used. Analogously, the number of new equations generated by one interaction rule in the lightweight calculus varies. Reducing one active pair may yield an arbitrarily large net, or any number of equations in the lightweight calculus. GPU-based algorithms usually have a fixed output size for every input.

Solving these challenges is by far not completed. In fact, we will see that dealing with these issues results in the performance deficits of the current implementation, cf. Section 5.8.

5.7.3 Overview of the Implementation

In this subsection, we describe the basic concept behind *ingpu*. As with *inets*, we base *ingpu* on the lightweight calculus. We represent agents and variables by a *unique id*, a *name*, an *arity* and a *list of ids* of the agents connected to an agent's auxiliary ports. Currently, agents and variables are simply distinguished by upper and lower case names. In fact, the name of a variable is not important: its identification by the unique id is sufficient for all computations. Naturally, we represent equations as pairs of agents. A configuration is simply a vector of equations - we leave the interface of the net implicit.

Control Flow

The basic control flow of our evaluator is simple. Specifically, our approach relies on a property of the reduction rules of the lightweight calculus (see Definition 2.4.1.2): substitutions done by $\xrightarrow{sub}$ and $\xrightarrow{col}$ never yield a new active pair/equation. This means that we can reach a normal form of a net (i.e., free of active pairs) by using only $\xrightarrow{int}$ and $\xrightarrow{com}$ rules.

Theorem 5.7.3.1 ([19]). *If $C_1 \longrightarrow^* C_2$ then there is a configuration C such that $C_1 \longrightarrow^* C \xrightarrow{sub}^* . \xrightarrow{col}^* C_2$, and C_1 is reduced to C applying only communication and interaction rules.*

In the lightweight calculus, the lion's share of the computation is done by $\xrightarrow{int}$ and $\xrightarrow{com}$. The former reduces expressions (i.e., equations/active pairs) by replacing them with the RHS of the corresponding interaction rule. The latter generates new active equations that can be reduced by $\xrightarrow{int}$. The collect and substitution steps are in a sense cosmetic: both resolve variables in order to provide a better readable form of the result net. They perform no "actual" computation, i.e., rewriting of the graph represented by the set of equations. Hence, all $\xrightarrow{sub}$ and $\xrightarrow{col}$ steps can be pushed to the end of the computation. Therefore, *ingpu* performs parallel interaction and parallel communication in a loop until no more active pairs exist. For the remainder of this section, we will describe the implementation of the data-parallel versions of $\xrightarrow{int}$ and $\xrightarrow{com}$.

5.7.4 Parallel Interaction

The *interaction step* $\xrightarrow{int}$ can be parallelized in a straightforward way. Clearly, $\xrightarrow{int}$ fits the SIMD model very well: the same program (i.e., the set of interaction rules) is applied to each active pair, and replacing a pair is completely independent of all others.

The problematic part about the $\xrightarrow{int}$ step is the varying output size of an active pair reduction, which we mentioned in Section 5.7.2. In general, a rule RHS may contain an arbitrary number of equations. Unfortunately, *Thrust*'s algorithms can only return a fixed number of results per individual input. This means that it is not feasible to return a dynamically-sized list of equations as the result of an interaction. For the time being, we have solved this problem in a pragmatic way by setting a maximum number *n* of equations per RHS. Applying any interaction rule must then yield exactly n equations. Between zero and $m \leq n$ equations represent the actual result. The remaining $n - m$ equations are *dummies*, resulting in a fixed result size for each application of the kernel.

Obviously, the dummy equations must be filtered in a subsequent computation step. Fortunately, *Thrust* provides the `remove()` function, which is a parallel version of Haskell's `filter`. Figure 5.2 illustrates the full interaction step: every equation in the input array that represents an active pair is reduced, i.e., the n result slots ($n = 3$ in the figure) are filled with the equations of the RHS of the corresponding rule and dummy equations (empty slot). Afterwards, the dummy equations are removed and the resulting equations are merged into a single array.

This approach is straightforward, but has performance drawbacks. The filter and merge operations on the result arrays are up to several hundred times slower than the actual parallel interaction step. This contributes to the current inefficiency of *ingpu*, which we will discuss in detail in Section 5.8.

5.7.5 Parallel Communication

After the interaction phase above has completed, we apply an algorithm corresponding to the *communication* rule to generate new active pairs. Recall the mechanics of $\xrightarrow{com}$: communication needs to find two equations $\{x = t, x = s\}$ where x is a variable and s, t are terms

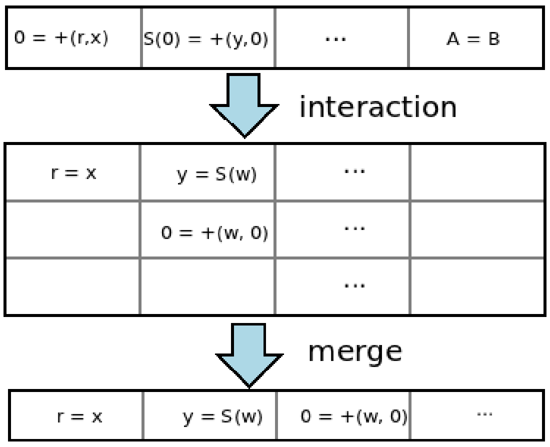


Figure 5.2: The interaction step

and merge them to a single equation $\{s = t\}$. Let us call two such equations sharing a variable *communication-eligible*. This is harder to parallelize than $\xrightarrow{int}$: we need to find eligible pairs of equations first before we reduce them. Unfortunately, this is where we run into a problem mentioned in Section 5.7.2: our net is only represented as an array of independent equations. There is no additional pointer structure between them to represent connections (cf. the `Agent` structure in Section 5.7.3). Therefore, we currently perform the following (inefficient) procedure to find communication-eligible pairs of equations: we sort all equations of the pattern $x = t$ by the id of the variable x. This places communication-eligible pairs of equations next to each other in the array (since the same variable occurs in them). We can now proceed to merge these pairs into single equations. We do this by using *Thrust*'s `reduce_by_key()`: this function essentially works like Haskell's `fold` with an added predicate function. Only adjacent array elements that satisfy the predicate are folded. For example, consider a list of numbers $\{2, 0, 3, 3, 3, 7, 5, 5\}$: calling `reduce_by_key()` on this list with addition as folding function and equality as predicate yields the result $\{2, 0, 9, 7, 10\}$.

Hence, we consider two equations that share a common variable as equal. This way, we merge communication-eligible pairs and leave all other equations untouched. The communication step can be summed up by the following pseudo-code:

```
let E be the array of all equations of shape x=t
sort E by the id of left part of each equation
reduce_by_key(E) with predicate p and reduce functor r
  where
  p (x,t) (y,s) = x==y
  r (x,t) (y,s) = (s,t)
```

Thrust provides efficient implementations of parallel sorting algorithms, but sorting and reducing the equations still represents a major performance bottleneck for large inputs. We will discuss this issue in detail in Section 5.8. For the remainder of this section, we will show that our algorithm is correct.

5.7.6 Correctness of the Algorithm

The communication algorithm seems fairly straightforward, but we have not discussed one important point: what if an equation consists of two variables, e.g. $x = t$ where t is a variable? Which one is used for comparison in the sorting of the equations? Surprisingly, it does not matter for our algorithm. We will now show that it is correct in the sense that any possible active pair that can be generated by $\xrightarrow{com}$ in the lightweight calculus will also be generated by *ingpu*.

First however, we highlight that some active pairs cannot be generated by the communication algorithm described above. Consider a set of equations $\{A = x, x = y, y = B\}$. Clearly, both x and y can be resolved, yielding the active pair $A = B$. However, we need to perform two consecutive $\xrightarrow{com}$ steps, first eliminating x and then y or the other way round.[2] The communication algorithm above would only perform one of these communications, depending on whether x or y was used for sorting. The result would be $\{A = y, y = B\}$ (if x was reduced), which cannot be used in the interaction phase. A second pass of the communication algorithm would be necessary to yield the active equation $A = B$. The following pseudo-code of the evaluation function reflects this:

```
equation_list evaluate(equation_list L) {
   transfer L to device (GPU) memory
   do {
      perform interaction as in Section 5.7.4
      perform communication as in Section 5.7.5
   }
   while (at least one interaction or one communication
      has occurred in the previous loop)
   transfer L back to host (CPU) memory
   return L
}
```

This way, if the communication algorithm does not yield any active pairs, it is performed again until it either generates new active pairs or no longer performs any $\xrightarrow{com}$ steps. In the latter case, the program terminates. We can show that the algorithm is correct in the sense that it generates all possible active pairs and evaluates them.

Proposition 5.7.7 (Correctness of `evaluate()`). *Let E be a set of equations. Then, $R =$ `evaluate(`E`)` contains no active pairs, and no further active pairs can be generated by applying $\xrightarrow{com}$ in R.*

Proof. We show that for any set of equations that can potentially yield an active pair, `evaluate()` will generate this active pair. Such a set of equations has the general form $\{A = x_1,\ x_1 =$

[2]Recall that the order of reduction steps in the lightweight calculus does not influence the final result due to uniform confluence (see Proposition 2.4.1.4).

$x_2,\ x_2 = x_3,\ \ldots,\ x_{i-1} = x_i,\ x_i = B\}$. Applying the communication algorithm will result in at least one $\xrightarrow{com}$ step, no matter which of the variables of each two-variable equation is used for sorting. Hence, the size of the set of equations decreases by at least 1 in each iteration of the loop in `evaluate()`. After at most i loops, the communication algorithm will yield $\{A = B\}$, which can then be reduced in the subsequent interaction phase of `evaluate()`. □

Remark Note that the algorithm may not be able to apply all $\xrightarrow{com}$ steps to sets of equations that potentially do *not* yield an active pair. For example, $\{A = x_1,\ x_1 = x_2,\ x_2 = x_3,\ \ldots,\ x_{i-1} = x_i,\ x_i = y\}$ may not reduce to $\{A = y\}$ depending on the choice of comparison variables for sorting. However, these $\xrightarrow{com}$ steps do not yield any active pairs and hence are of secondary importance, much like $\xrightarrow{col}$ and $\xrightarrow{sub}$ (c.f. Theorem 5.7.3.1).

5.8 Benchmarks and Future Improvements

In this section, we present some benchmark results and discuss possible performance improvements. First, we compare *ingpu* to existing, sequential interaction nets evaluators. Second, we identify performance bottlenecks and discuss potential solutions.

5.8.1 Performance Comparison

Our implementation is still considered experimental. Unfortunately, *ingpu* currently performs slower than the more mature sequential evaluators *inets* [23] and *amineLight* [19] in most cases. However, we can identify which parts of *ingpu* are slow and which ones are fast. Consider the benchmark comparison in Figure 5.3. We ran our tests on a machine with an Intel Core Duo 2.2Ghz CPU, 2GB of RAM and a Geforce GTX 460 graphics card.

time in seconds	amineLight	inets	ingpu
Ackermann(3,7)	0.4	0.59	30.4
Fibonacci(20)	0.03	0.043	10.5
L-System(27)	1.13	1.49	1.28

Figure 5.3: Benchmark results

The code for the rule definitions of the benchmarks can be found in the appendix. On standard benchmark functions like *Ackermann* or *Fibonacci*, *ingpu* performs very poorly. The sequential implementations are much faster here. However, the result on the third benchmark *L-System* is much more competitive, at least outperforming *inets*. This set of rules computes the nth iteration of a basic L-System that models the growth of Algae [52], given by the following rewrite System:

$$A \to AB \qquad\qquad B \to A$$

So why is *ingpu* so much faster for the *L-System* ruleset than for *Ackermann* and *Fibonacci*? Two main reasons can be given, which at the same time highlight the most glaring performance problems:

Inefficient communication Profiling shows that for *Ackermann*, almost all of the execution time of *ingpu* is spent in the communication phase and the merging part of the interaction phase. The actual interaction kernel (replacing equations) runs very fast, taking less than 0.1 percent of the total time.

Available parallelism The term *available parallelism* was coined in [41], and refers to the number of expressions that can be evaluated in parallel at a given time during the execution of a program. In our case, this is simply the number of active pairs that exist at the same time at a certain iteration of *ingpu*'s main loop. A high number of active pairs fits the GPU programming model well, as GPUs are optimized to handle hundreds of thousands of data-parallel inputs. Conversely, a low amount of parallelism is not sufficient to leverage the full computing power of the GPU. Figure 5.4 shows the number of parallel interactions per loop for *Ackermann* and *L-System*. While *Ackerman* has less than 200 concurrent active pairs for the majority of its execution time, *L-System* performs several hundreds of thousands of parallel interactions towards the end of the computation. Moreover, *L-System* performs much fewer loop iterations overall (50 vs. 11000). This means that much less time is spent in the slow communication phase in proportion to the total number of interactions.

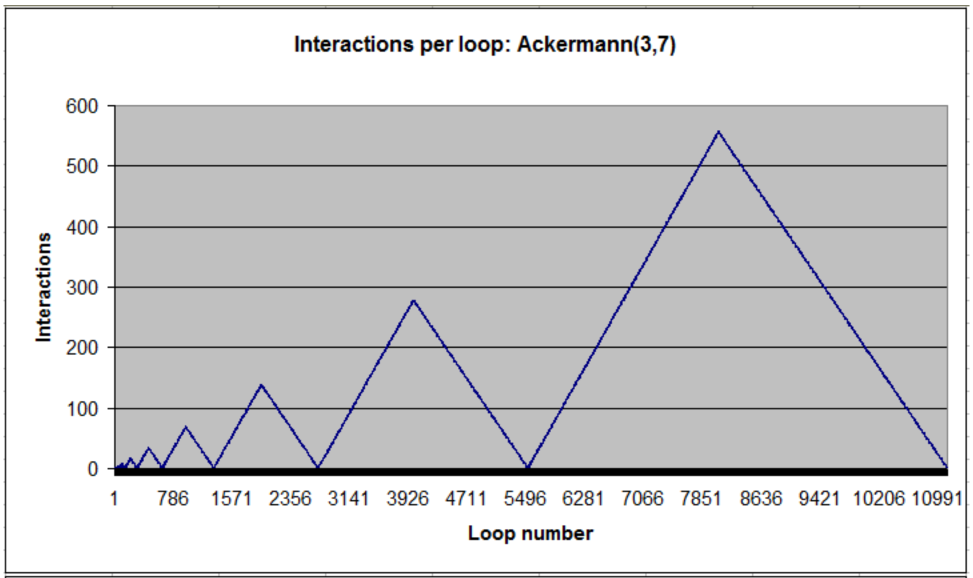
Figure 5.4 also gives some insight on the dynamics of available parallelism in our benchmarks. Interestingly, the number of concurrent active pairs for *Ackermann* repeatedly decreases and increases in a quasi oscillating fashion. This results in the rather low average number of active pairs per loop. In contrast, the available parallelism in the execution of *L-System* strongly increases, as is expected considering the exponential growth of the L-System. The slight drops in parallelism are the result of duplicating the parameter for the number of iterations in every loop.

5.9 Performance Improvements

Our initial design of *ingpu* can be considered experimental. Various small and big improvements can be made to increase its overall performance. In particular, we have identified a few optimizations that have the potential for a considerable performance increase. In this section, we describe these performance improvements to *ingpu*.

5.9.1 Improved Net Representation

Both interaction and communication can benefit from a better net representation in the GPU memory. While the "array of equations" approach closely follows the theoretical definition of the interaction nets calculus, it is quite slow: the lack of a more sophisticated pointer structure makes a sorting pass in the communication phase necessary.



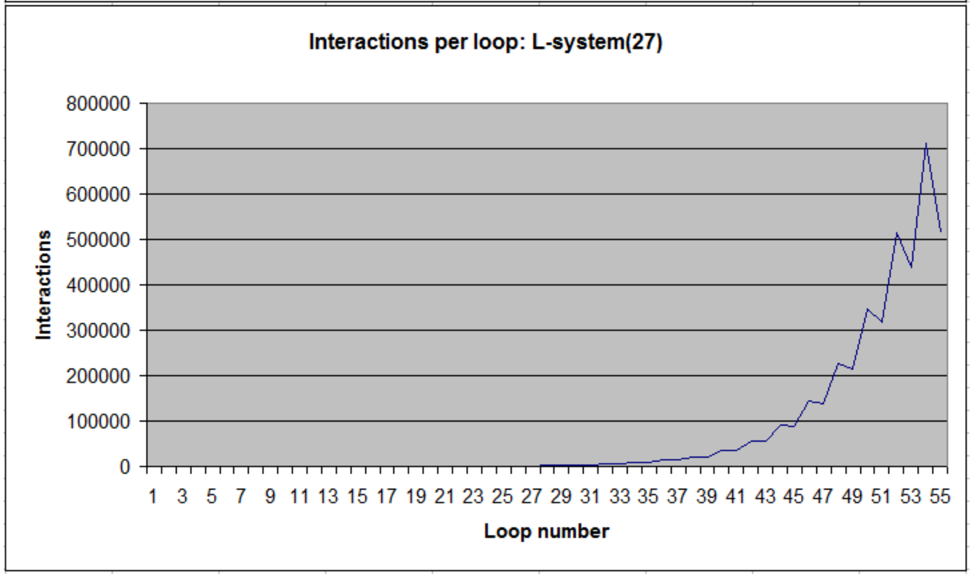


Figure 5.4: Available Parallelism of *Ackermann* and *L-System*: the x-axis shows the loop number, the y-axis the number of parallel interactions.

Instead of an array of equations, an interaction net can be represented more efficiently in the device memory. We store all agents and variables of the net in an array called the *agent table*. The *unique id* of each agent/variable determines the position in the array (i.e., the agent with id n is stored at array position n). The agent table also stores all non-active pair connections in the net, i.e., all connections/equations that include a variable. Active pairs are stored in a different array as a pair of the unique ids of the respective agents.

This representation has two main advantages. First, any agent or variable can be accessed in constant time via their id. This makes a sorting pass in the communication phase unnecessary. In addition, the interaction phase can also be improved by constant-time agent access. Second, the size of the data-parallel input for both phases is reduced considerably. Previously, the whole array of equations (i.e., the complete net) was passed to the kernels. Now, we simply pass the list of active pair ids as well as a pointer to the agent table to the interaction kernels. Similarly, the communication kernels only process the equations that were newly created by the preceding interaction.

5.9.2 Interaction: Efficient Handling of Dynamic Output Size

As we discussed in Section 5.8, one performance bottleneck is the merging of result arrays, which comes from the varying output size (the number of equations) of interaction rules. The arbitrary size of a rule RHS is a crucial property of interaction nets and cannot be removed. However, we can handle output of dynamic size more efficiently by exploiting information about each interaction rule: the RHS size of each rule is known and constant. Therefore, we can adapt a so-called *scan-based* approach. A *scan* is a function that takes an array of values $[a_1, \ldots, a_n]$ and an associative function $*$ and returns an array $[a_1,\ a_1 * a_2,\ \ldots,\ a_1 * a_2 * \ldots * a_n]$. The nth element of the result is the *-product of the first n elements of the input array. A scan can be parallelized very well on the GPU and is the key to handling dynamic output size more efficiently.

In [33], the authors describe an efficient way to compute and render L-systems (a particular kind of rewriting system) on the GPU. The problem of dynamic output size is solved by splitting the parallel rewriting step in *three* passes:

count for each input, the size of the output is computed in parallel.

scan perform a scan on the result array of the count pass. This yields an index for each of the original inputs that can be used to allocate the individual outputs on a single output array.

rewrite perform the actual computation on the original input, and store the output at the output array index determined by the scan pass.

Initial tests show that the *count* and *scan* passes are considerably faster than the original way of removing duplicates. The rewrite pass modifies the agent table by adding the new agents of the rule RHS. Due to the uniform confluence property of interaction nets, all changes from all active pairs can be applied to the agent table at the same time: each active pair affects a different part of the table.

5.9.3 Improved Communication

The previously discussed communication algorithm is very slow. Due to the fact that an interaction net is represented as a list of independent equations, we have to sort the complete list repeatedly to find communication-eligible equations. However, it is possible to reduce this search space: only equations that are directly connected to a given active pair(i.e., its interface) are potentially communication-eligible. We gain a speedup by considering this subset of equations only.

As with the interaction phase, the improved net representation allows for constant-time lookup of variables. This makes the sorting of equations in the original communication phase obsolete.

While active pairs are the inputs of the interaction phase, the communication phase evaluates the equations resulting from the active pair reduction. The main challenge is now to determine which equations yield a new active pair (in combination with the net stored in the agent table) and which merely update the agent table. In particular, an active pair could result from communicating several variables (see Section 5.7.6). Therefore, we also split communication into several passes, again employing a *scan approach*. First, we determine which variables/equations give rise to a new active pair and store this information in an array. Second, we again scan the array to compute the number of new active pairs and their positions in the final output array. Finally, we perform the actual communication for the determined variables. This multi-pass scan approach can again yield a better performance than the original implementation.

Updated Benchmark Results

Unfortunately, even with these improvements *ingpu* currently fails to match the performance of existing implementations on the benchmarks with low parallelism (Ackermann and Fibonacci). However, for the L-system ruleset, *ingpu* now performs faster than the other systems. The following benchmarks were conducted on a machine with a Intel i5 3.1Ghz CPU, 8GB of RAM and a Geforce GTX 670 GPU.

time in seconds	amineLight	inets	ingpu
Ackermann(3,7)	0.2	0.29	11.5
Fibonacci(20)	0.02	0.03	2.48
L-System(27)	0.48	0.66	0.19

Figure 5.5: Updated Benchmarks

Ackermann and *Fibonacci* do get a decent speedup, but are still 20-100 times slower than the sequential implementations. The *L-System* benchmark manages to beat the sequential systems by a factor of 2.5. This result can be explained by looking at the average available parallelism for each test:

Note the direct correlation between the average number of concurrent active pairs and the slowdown of *ingpu*. It seems that the available parallelism (see Section 5.8) is indeed a decisive factor for the viability of a GPU-based implementation.

	avg. active pairs per loop	ingpu vs amineLight slowdown
Ackermann(3,7)	200	60
Ackermann(3,8)	400	30
Fibonacci(20)	30	134
L-System(27)	67000	0.39

Figure 5.6: Available Parallelism vs. *ingpu* Performance

5.10 Parallel Evaluation with *ingpu*: Summary

We have presented a novel approach for implementing interaction nets evaluation: By using graphics cards (GPUs) as target platform, we can realize the full theoretical parallelism potential of interaction nets: all concurrent active pairs are evaluated at the same time, in parallel. While the current state of the work does not outperform existing systems for examples with comparatively low parallelism, we have shown that our approach is a promising one if the degree of parallelism is high.

CHAPTER 6

Conclusion

In this chapter we conclude the thesis. We summarize our results and discuss open problems as well as future research directions. In addition, we compare our achievements with related work.

6.1 Related Work

6.1.1 Generic Rules and the Lightweight Calculus

The textual calculus for interaction nets was initially defined in [11]. We based our extensions on the improved *lightweight* calculus, which was introduced in [19]. A different approach to higher-order computation in the interaction calculus inspired by combinatory reduction systems [29] can be found in [14].

Besides *inets*, several implementations of interaction nets evaluators exist. Examples are *amineLight* [19] or INblobs [2]. To the best of our knowledge, none of these systems support generic interaction rules.

The extension of interaction nets to a practically usable programming language has been the topic of several publications. For example, in [40] the authors propose a way to represent higher-order recursive functions like *fold* or *unfold*. A different approach to more complex rule patterns can be found in [48].

6.1.2 Graph Reduction Systems

Portgraph systems [4] provide an alternative semantics for interaction nets. They define a reduction rule as a single graph where the LHS and RHS are connected by a special reduction node (representing the arrow $\Rightarrow$). Interaction nets can be seen as a restricted form of portgraph systems satisfying uniform confluence. General portgraph systems may be non-confluent or non-deterministic. For example, they have been used to model biological systems with probabilistic rewriting rules. A recent implementation of portgraph systems is PORGY [3], which can be used to analyse and evaluate graph reduction systems (including interaction nets) with a focus on evaluation strategies.

Besides interaction nets, there are several formal models based on graph rewriting (or reduction). One example is *Graph Programs (GP)* [45], a graph based programming language with a graphical notation for reduction rules, quite similar to interaction nets. However, the reduction rules of GP are less constrained, allowing for a large variety of graphs on the LHS. As a consequence, functions in the GP language can be non-deterministic, and may hence not satisfy any form of confluence. Graph Programs have been applied to different problems such as defining algorithms on graphs [46] and minimizing finite automata [47].

6.1.3 Irregular Algorithms and *ingpu*

Evaluation of interaction nets can be considered an *irregular algorithm*, in the sense that it operates on a pointer structure (a graph) rather than a dense array. Bridging the conceptual gap between the irregular nature of interaction nets and the dense structure of typical GPU programs (e.g., dense matrix operations) is strongly related to the implementation challenges discussed in this paper. We have been inspired by the insights of parallelizing irregular algorithms in [41] when implementing *ingpu*. We also borrowed the term *available parallelism* from this work.

The efficient parallelization of general graph algorithms using GPUs has been the topic of several publications (for example, [22]). As part of future work, we plan to use these insights to achieve a better representation of interaction nets on the GPU.

The previous subsection mentioned several implementations of interaction nets. However, only few works (for example, [43]) on parallel evaluation of interaction nets exist. This is surprising, considering their potential for parallelism. To the best of our knowledge, there is no previous work on a GPU-based implementation.

With regard to functional programming on the GPU in general, two recent systems are *Obsidian* [49] and *Accelerate* [6], both being extensions of Haskell.

6.1.4 Monads and Side Effects

Our notion of IN monads is based on their application to functional programming [26, 42, 51]. However, it might be interesting to try a more categorial approach to monads in interaction nets. In [7], the author uses notions from category theory to explicitly define interaction rule application and rewriting of nets. Besides in functional programming, monads have been used in several other domains. A recent application can be found in [27], where monads serve for structuring mechanisms in interactive theorem provers.

As we mentioned in Section 2.6.2, monads are not the only approach to side effects. In particular, approaches based on *linear logic* [1, 30] could be employed to handle impure functions in interaction nets: there is a strong relation between both formalisms (cf. Section 2.6.1).

6.2 Open Problems

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 [40] may be a possible direction to achieve this. In particular, this could help us to more conveniently define monads that are based on other monads, such as Monad Transformers and Tracing of Computation [44].

As we discussed in Section 5.8, the performance of *ingpu* is not satisfying, in particular for examples with a low degree of parallelism. While the design changes of Section 5.9 yield considerable improvements for benchmarks with high parallelism, the former still perform worse than existing sequential implementations. It remains to be seen whether low-parallelism programs on the GPU can outperform sequential evaluators. It would be interesting to identify minimal degrees of parallelism in irregular algorithms that make a GPU implementation worthwhile.

The most recent hardware generation of NVIDIA GPUs directly supports dynamic parallelism: kernels can launch other kernels themselves, independently of the CPU. This could be an interesting feature for *ingpu*, and will be investigated as part of future work. At the time of writing, the API support for dynamic parallelism is still experimental.

6.2.1 Generic Rules and Self-Interaction

In Section 3.1.1, we emphasized that generic rules do not allow self-interaction: a generic rule $\alpha \bowtie B$ does not match the active pair $B \sim B$. The reason for this is that the RHS of a generic rule may not satisfy Lafont's *symmetry* property of the *no ambiguity* constraint (see Section 2.2). Consider the ret-rules in the IN monads of Chapter 4. Each ret-rule is inherently asymmetric: the active pair $ret \sim ret$ is ambiguous as interaction nets are not oriented. In contrast, the valid Haskell expression **`return return`** contains a distinction between the calling function (the first **`return`**) and its argument (the second **`return`**).

On the other hand, the generic rules for δ and ϵ have symmetric RHSes, which makes self-interaction feasible. Intuitively, this makes sense as there is no orientation: two ϵ-agents erase each other, there is no distinction or hierarchy between them.

Instead of completely forbidding self-interaction for generic rules, one could allow self-interaction for rules whose RHSes satisfy the symmetry constraint (e.g, the rules for δ and ϵ) and forbid it for all others (ret, $>>=$). However, this seems like a less clean solution than simply ruling out self-interaction. Similarly, it does not make sense to enforce RHS symmetry for generic rules if we consider the IN monad rules for ret and $>>=$. In summary, it seems that the problem of generic rules and self-interaction cannot be easily solved, except by forbidding self-interaction and adding ordinary self-interacting rules (under the DPC).

6.3 Summary

In this thesis, we set out to extend the formal model of computation interaction nets. Our goal was to bring interaction nets closer to a practically usable programming language that could benefit from their formal properties.

While several open questions and problems remain (as discussed in the previous subsection), we have made substantial progress in two aspects. First, our extension of the formal model towards practical usability, and second, the implementation of these extensions, including an investigation of maximally parallel evaluation of interaction nets. Our results are an important step towards "real world interaction nets programming". However, many further improvements need to be be made to achieve this goal (e.g., providing a more efficient implementation of parallel evaluation).

6.3.1 Generic Rules and their Application to Side Effects

As the main theoretical result, we presented generic rules for interaction nets (Chapter 3). These rules are substantially more powerful than ordinary interaction rules and allow for a more general pattern matching. This adds a higher-order character to interaction rules, including a way to model partially evaluated functions. This extension is conservative: using appropriate constraints, we ensure that the reduction relation satisfies uniform confluence. Our theoretical results are a substantial step towards promoting interaction nets to a practically usable programming language. In combination with nested patterns, generic rules shift interaction nets away from their strongly constrained, assembly-like programming style towards practical usability. Our rule type system ensures that the matching of generic rules is consistent with the type restrictions of the monadic operators. While being sufficient for this task, the system is fairly simple and can of course be refined. This will be subject of future work.

We have shown that individual monads can be defined using generic rules, realizing various computational side effects directly in interaction nets. Several examples can be found in the IN Monads of Chapter 4. Generic rules allow for elegant and concise rule definitions that are very similar to corresponding functional programs. In particular, there is no need for encodings of the lambda calculus and explicit function application [36] or externally defined programs [13]. However, the former may of course be combined with our approach, as in the example in Section 4.4.

In addition, we extended *inets* with interaction rules with nested patterns. Nested patterns, particularly in combination with generic rules, allow for a concise definition of complex functions. We presented an algorithm for the translation of nested patterns to ordinary ones, showed its correctness and discussed its performance.

6.3.2 The Lightweight Calculus: Theory and Implementation

We extended the interaction nets calculus by generic rules. This extension provides an alternative precise semantics to the graphical notation of generic interaction rules. This is particularly important for variadic rules, which use an intuitive dot notation to express an arbitrary number of ports in the graphical setting. Our approach using variadic names and ranges is concise and precisely formulates the mechanics of the dot notation of the graphical rewrite rules.

The lightweight calculus is an important foundation of interaction nets based programming languages. 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.

6.3.3 Parallel Evaluation with *ingpu*

In Chapter 4, we have presented ongoing work on *ingpu*, a GPU-based evaluator for interaction nets. This is a novel approach that heavily utilizes their potential for parallelism: all active pairs that are available at the same time are evaluated in parallel. Previous evaluators are sequential or only allow a fixed number of concurrent interactions (e.g., capped by the number of cores of the CPU). A GPU with hundreds of cores is better suited to perform a high number of small computations (i.e., reductions of active pairs). Still, the implementation poses a challenge due to the dynamic nature of interaction nets evaluation and the restrictions of the GPU computing model.

The work-in-progress status of *ingpu* is clearly visible in the benchmark results of Section 5.8. While parallel evaluation is generally expected to be faster than a sequential one, our current implementation mostly performs weaker than existing evaluators. However, we argue that the potential of *ingpu* can be seen in the difference between the individual results. In our *L-System* test, *ingpu* performs more than 50 times faster than for *Ackermann*, in terms of interactions per second. The major part of the slowdown is caused by the communication phase, which should be seen as an intermediate solution. In contrast to this, the interaction phase (parallel reduction of the active pairs) is very fast and shows that interaction nets and GPU are a promising match.

For current and future work, we plan to optimize the system to improve the obvious performance bottlenecks. Initial tests show that with a more efficient net representation in GPU memory and the removal of result array merging (see Section 5.9), *ingpu* outperforms CPU-based systems at least for highly parallel benchmarks.

6.3.4 Outlook

Despite the recent advances of interaction nets, it remains to be seen if the model can provide a revolutionary new approach to programming languages in practice. At the very least, the current trend towards massive parallel and concurrent programming calls for new models of computation that focus on these properties. The future will show if and when inherently sequential models like the Turing machine may be replaced by newer approaches. Interaction nets could be a suitable candidate.

APPENDIX A

Appendix

A.1 Benchmark Rulesets of Section 5.8

The following rulesets have been used in the benchmarks in Section 5.8:
Ackermann:

```
A1(y,r)>< Z => r~S(y);
A1(y,r)>< S(x) => A2(S(x),r)~y;
A2(x,r)>< Z => Pred(A1(S(Z),r))~x;
A2(x,r)>< S(y) => x~Dup(Pred(A1(w2,r)),A1(y,w2));
Dup(a1,a2)>< Z => a1~Z, a2~Z;
Dup(a1,a2)>< S(x) => a1~S(w1), a2~S(w2), x~Dup(w1,w2);
Pred(a1)>< S(x) => a1~x;
```

Fibonacci:

```
Fib(r) >< Z => r~Z;
Fib(r) >< S(x) => FibH(r)~x;
FibH(r) >< Z => r~S(Z);
FibH(r) >< S(x) => Add(r, s) ~ t, Fib(s)~S(d1), Fib(t)~d2, Dup(d1,d2) ~ x;
Add(r,x) >< Z => r~x;
Add(r,x) >< S(y) => r~S(w), Add(w,x)~y;
Dup(d1, d2) >< Z => d1~Z, d2~Z;
Dup(d1, d2) >< S(x) => d1~S(a), d2~S(b), Dup(a,b)~x;
```

L-System:

```
L(r,i) >< A(s) => La(r,s)~i;
L(r,i) >< B(s) => Lb(r,s)~i;
La(r,s) >< Z   => r~A(s);
Lb(r,s) >< Z   => r~B(s);
La(r,s) >< S(x) => L(r,d)~A(y), L(y,e)~B(s), Dup(d,e)~x;
```

```
Lb(r,s) >< S(x) => L(r,x)~A(s);
Dup(d1, d2) >< Z => d1~Z, d2~Z;
Dup(d1, d2) >< S(x) => d1~S(a), d2~S(b), Dup(a,b)~x;
```

Bibliography

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

[2] Jose Bacelar Almeida, Jorge Sousa Pinto, and Miguel Vilaca. A tool for programming with interaction nets. *Electronic Notes in Theoretical Computer Science*, 219:83–96, 2008.

[3] Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. Porgy: Strategy-driven interactive transformation of graphs. In Rachid Echahed, editor, *TERMGRAPH*, volume 48 of *EPTCS*, pages 54–68, 2011.

[4] Oana Andrei and Hélène Kirchner. A rewriting calculus for multigraphs with ports. *Electr. Notes Theor. Comput. Sci.*, 219:67–82, 2008.

[5] 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.

[6] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In Manuel Carro and John H. Reppy, editors, *DAMP*, pages 3–14. ACM, 2011.

[7] Marc de Falco. An explicit framework for interaction nets. *Logical Methods in Computer Science*, 6(4), 2010.

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

[9] Maribel Fernández and Ian Mackie. From term rewriting to generalised interaction nets. In Herbert Kuchen and S. Doaitse Swierstra, editors, *PLILP*, volume 1140 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 1996.

[10] Maribel Fernández and Ian Mackie. Interaction nets and term-rewriting systems. *Theor. Comput. Sci.*, 190(1):3–39, 1998.

[11] Maribel Fernández and Ian Mackie. A calculus for interaction nets. In Gopalan Nadathur, editor, *PPDP*, volume 1702 of *LNCS*, pages 170–187. Springer, 1999.

[12] Maribel Fernández and Ian Mackie. Operational equivalence for interaction nets. *Theor. Comput. Sci.*, 297(1-3):157–181, 2003.

[13] 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.

[14] Maribel Fernández, Ian Mackie, and Jorge Sousa Pinto. A higher-order calculus for graph transformation. *Electr. Notes Theor. Comput. Sci.*, 72(1):45–58, 2007.

[15] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[16] ingpu. `https://github.com/euschn/ingpu` [Online, accessed 30-August-2012].

[17] 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.

[18] Abubakar Hassan, Ian Mackie, and Shinya Sato. Compilation of interaction nets. *Electr. Notes Theor. Comput. Sci.*, 253(4):73–90, 2009.

[19] Abubakar Hassan, Ian Mackie, and Shinya Sato. A lightweight abstract machine for interaction nets. *ECEASST*, 29, 2010.

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

[21] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[22] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In Calin Cascaval and Pen-Chung Yew, editors, *PPOPP*, pages 267–276. ACM, 2011.

[23] The inets project site [online, accessed 13-april-2012]. `http://gna.org/projects/inets`.

[24] Eugen Jiresch. Realizing impure functions in interaction nets., 2011. *ECEASST 38*.

[25] Eugen Jiresch and Bernhard Gramlich. Realizing monads in interaction nets via generic typed rules . `http://www.logic.at/staff/gramlich/papers/techrep-e1852-2011-01.pdf`, 2011. Technical Report E1852-2011-01.

[26] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In Mary S. Van Deusen and Bernard Lang, editors, *POPL*, pages 71–84. ACM Press, 1993.

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

[28] J.-W. Klop. *Term Rewriting Systems*, volume 2, chapter 1, pages 2–117. Oxford University Press, 1992.

[29] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theor. Comput. Sci.*, 121(1&2):279–308, 1993.

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

[31] Yves Lafont. From proof-nets to interaction nets. *Advances in Linear Logic*, London Mathematical Society Lecture Note Series(222):225–247, 1995.

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

[33] Markus Lipp, Peter Wonka, and Michael Wimmer. Parallel generation of multiple l-systems. *Computers & Graphics*, 34(5):585–593, 2010.

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

[35] Ian Mackie. Interaction nets for linear logic. *Theor. Comput. Sci.*, 247(1-2):83–140, 2000.

[36] Ian Mackie. Efficient lambda-evaluation with interaction nets. In Vincent van Oostrom, editor, *RTA*, volume 3091 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2004.

[37] Ian Mackie. Encoding strategies in the lambda calculus with interaction nets. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *IFL*, volume 4015 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2005.

[38] Ian Mackie. A visual model of computation. In Jan Kratochvíl, Angsheng Li, Jirí Fiala, and Petr Kolman, editors, *TAMC*, volume 6108 of *Lecture Notes in Computer Science*, pages 350–360. Springer, 2010.

[39] Ian Mackie, Abubakar Hassan, and Shinya Sato. Interaction nets: programming language design and implementation. *Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques*, 2008.

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

[41] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, Muhammad Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-driven optimizations for amorphous data-parallel programs. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, *PPOPP*, pages 3–14. ACM, 2010.

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

[43] Jorge Sousa Pinto. Parallel evaluation of interaction nets with mpine. In Aart Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 353–356. Springer, 2001.

[44] Maciej Piróg and Jeremy Gibbons. Tracing monadic computations and representing effects. In James Chapman and Paul Blain Levy, editors, *MSFP*, volume 76 of *EPTCS*, pages 90–111, 2012.

[45] Detlef Plump. The graph programming language gp. In Symeon Bozapalidis and George Rahonis, editors, *CAI*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer, 2009.

[46] Detlef Plump and Sandra Steinert. Towards graph programs for graph algorithms. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *ICGT*, volume 3256 of *Lecture Notes in Computer Science*, pages 128–143. Springer, 2004.

[47] Detlef Plump, Robin Suri, and Ambuj Singh. Minimizing finite automata with graph programs. *ECEASST*, 39, 2011.

[48] François-Régis Sinot and Ian Mackie. Macros for interaction nets: A conservative extension of interaction nets. *Electr. Notes Theor. Comput. Sci.*, 127(5):153–169, 2005.

[49] Joel Svensson, Koen Claessen, and Mary Sheeran. Gpgpu kernel implementation and refinement using obsidian. *Procedia CS*, 1(1):2065–2074, 2010.

[50] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

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

[52] Wikipedia. L-system — Wikipedia, the free encyclopedia [online, accessed 23-november-2011]. `http://en.wikipedia.org/wiki/L-system`, 2011.