University of Sussex

# Design and implementation of a low-level language for interaction nets

**Shinya Sato**

Submitted for the degree of Doctor of Philosophy

University of Sussex

September 2014

UNIVERSITY OF SUSSEX

Shinya Sato, Doctor of Philosophy

# Design and implementation of a low-level language for interaction nets

## Summary

Interaction nets are a graphical model of computation based on a restricted form of graph rewriting. A specific net can represent a program with a user-defined set of nodes and computation is modelled by a user-defined set of rewrite rules. This very simple model has had great success in modelling sharing in computation (specifically in the lambda calculus), and there is potential for generating a new theoretical foundation of parallel computation since all computation steps are local and thus can be implemented in parallel.

This thesis is about the implementation of interaction nets. Specifically, for the first contributions we define a low-level language as an object language for the compilation of interaction nets. We study the efficiency and properties of different data structures, and focus on the management of the rewriting process which is usually hidden in the graph rewriting system. We provide experimental data comparing the different choices of data structures and select one for further development. For the compilation of nets and rules into this language, we show an optimisation such that allocated memory for agents is reused, and thus we obtain optimal efficiency for the rewriting process.

The second part of this thesis describes extensions of interaction nets so that they can be used as a programming language. Interaction nets in their pure form are quite restrictive in expressive power. By extending the notions of agents and rules we can express computation more naturally, yet still preserve the good properties (such as strong confluence) of the rewriting system. We then implement a selection of algorithms using and extending the compilation techniques developed in the first part of the thesis. We also demonstrate experimental results on multi-core CPUs, using the Posix-thread library, thus realising some of the potential for parallel implementation mentioned above.

# Acknowledgements

First, I am deeply grateful to my supervisor Ian Mackie, for his enormous support and encouragement. His insightful comments were also innumerably valuable during the course of my study.

My joint work with Abubaker Hassan on the INET project has greatly benefited this thesis. Discussions with Abubaker have been a great help to me. I also deeply appreciate his painstaking effort at proofreading this thesis.

Special thanks go to Maribel Fernández, Eugen Jiresch and Nikolaos Siafakas for their helpful opinions and information. I thank my friends from Sussex for their advice and encouragements.

My intellectual debt is to my late former supervisor Shinichi Yamada, and my former sub-supervisors Yasushi Kodama and Toru Sugimoto. I thank my fellow members of staff and administrators at Himeji Dokkyo University for providing me with an opportunity to join the INET project. Furthermore, I would particularly like to thank my examiners, Bernhard Reus and Jorge Sousa Pinto, for the fruitful discussions at the viva and for taking time to give invaluable feedback. Without their help this thesis would not have been possible.

I would also like to express my gratitude to my family for their moral support and warm encouragements.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Linear logic

Linear logic [21] has had a profound impact in computer science over the last 25-30 years. It has created new artefacts, such as proof nets, as well as providing deep insight into others. It has been applied to many diverse areas of computer science. Some examples include: security, complexity theory, semantics, compilation, type systems, etc. In many of these application domains, there are now dedicated workshops or special sessions in major conferences covering these topics.

It is not our goal to recall all the history of linear logic here, but we mention briefly two instances where linear logic has been used to give some spectacular results: optimal reduction in the $\lambda$-calculus and the full abstraction result for PCF through game semantics. Both of these results do not rely directly on linear logic, but both were found passing through it (often referred to as "looking through the linear logic looking glass").

For us, one of the key features of linear logic is the idea of proof nets: a graphical representation of proofs. They are fundamentally different from natural deduction, sequent calculus, or axiom presentations of logic, because they take away some of the sequential constraints of writing proofs. The multiplicative fragment of linear logic (tensor and par) works particularly well as proof nets, and notions of parallel proof work very well. It is this multiplicative fragment of proof nets that marked the starting point of interaction nets which is the topic of this thesis.

Proof nets are a graphical syntax for linear logic proofs and they offer a representation of proofs which are free from many of the permutation and commutation equivalences, and thus are better suited for the study of computation. Specifically, the sequent calculus presentation of linear logic lacks certain desirable properties:

- There is no notion of *canonical proof*. Logical rules can be permuted in a proof, which gives several representations of the same object.
- The cut elimination process is overly complicated by the need for commutation rules which essentially re-order the rules of the proof in order to create a principal cut and to eliminate it. Moreover, the cut-elimination steps are not deterministic; that is to say that in the right-hand side of the rewrite rule we are free to build the proof in one of several different ways by re-ordering the logical and structural rules.

For intuitionistic logic, natural deduction is a solution to these issues: there is a canonical notion of a proof, and a deterministic cut-elimination procedure. Proof nets can be seen as the corresponding proof structure for linear logic having such properties. Here is a well-known example, taken from Girard [22], of one of the problems of sequent calculus with respect to representation of proofs. Let $(r)$ and $(s)$ be two logical rules, and the cut is working on auxiliary formulas (not the main formula of the rules $r$ and $s$):

$$\dfrac{\dfrac{\Gamma, A}{\Gamma', \mathbf{A}}\,(r) \qquad \dfrac{A^{\perp}, \Delta}{\mathbf{A}^{\perp}, \Delta'}\,(s)}{\Gamma', \Delta'}\,(\mathsf{Cut})$$

If we permute the cut rule up through $r$ and $s$, then there are two possible choices depending on whether we first permute through $s$ or through $r$. These two choices are represented by:

$$\dfrac{\dfrac{\Gamma, \mathbf{A} \qquad \mathbf{A}^{\perp}, \Delta}{\dfrac{\Gamma, \Delta}{\dfrac{\Gamma', \Delta}{\Gamma', \Delta'}\,(s)}\,(r)}\,(\mathsf{Cut})} \qquad \dfrac{\dfrac{\Gamma, \mathbf{A} \qquad \mathbf{A}^{\perp}, \Delta}{\dfrac{\Gamma, \Delta}{\dfrac{\Gamma, \Delta'}{\Gamma', \Delta'}\,(r)}\,(s)}\,(\mathsf{Cut})}$$

There is also no notion of canonical form for cut-free proofs, as the following example indicates. There are two alternative proofs, $\pi_1$ and $\pi_2$ of the formula:

$$A^{\perp}, C, A \otimes B^{\perp}, B \otimes C^{\perp}$$

which differ only by the order of combining the axiom links. $\pi_1$ is:

$$\dfrac{\dfrac{\dfrac{}{A^{\perp}, A}\,(\mathsf{Ax}) \qquad \dfrac{}{B^{\perp}, B}\,(\mathsf{Ax})}{A^{\perp}, B, A \otimes B^{\perp}}\,(\otimes) \qquad \dfrac{}{C^{\perp}, C}\,(\mathsf{Ax})}{A^{\perp}, C, A \otimes B^{\perp}, B \otimes C^{\perp}}\,(\otimes)$$

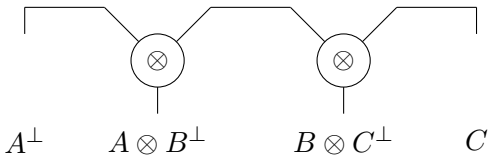
and $\pi_2$ is:

$$\dfrac{\dfrac{}{A^{\perp}, A}\,(\mathsf{Ax}) \qquad \dfrac{\dfrac{}{B^{\perp}, B}\,(\mathsf{Ax}) \qquad \dfrac{}{C^{\perp}, C}\,(\mathsf{Ax})}{C, B^{\perp}, B \otimes C^{\perp}}\,(\otimes)}{A^{\perp}, C, A \otimes B^{\perp}, B \otimes C^{\perp}}\,(\otimes)$$

Having several different proof objects, distinguished only by the syntax means that it is difficult to reason about equivalence (of proofs or programs).

The idea of proof nets is to define a proof structure that is free from these inessential permutations. This will then give us a calculus, in a similar spirit to the $\lambda$-calculus, that we can use for expressing proofs. Sequent calculus is often referred to as a sequential presentation of the logic, as one must select a given order on the rules. However, as we shall see below, we can free ourselves from such constraints. This is why proof nets are often referred to as the parallel syntax for proof theory.

The following proof net represents both $\pi_1$ and $\pi_2$ defined above. This justifies the motivation of finding a representation of proofs which factors out the order in which the rules were used to build the proof.




This graphical representation not only has advantages for representing proofs, but also for the cut-elimination procedure. We will not go further into this here, as we will focus on a generalisation of the idea: interaction nets.

## 1.2 Interaction nets

Lafont [36] introduced a paradigm in programming languages based on interaction nets—a networked system of interacting agents founded on proof nets for linear logic. From one perspective, they are a generalisation of proof nets because they allow user defined connectives and reduction rules (cut-elimination steps). We remark also that Bawden [9] has also considered a system of *connection graphs* that is very similar to interaction nets, and in particular he gave the first coding of a functional language (Scheme) into these networks before interaction nets were introduced.

Interaction nets are very appealing from a computational point of view. On the one hand we have a very simple graphical rewriting system which enjoys properties such as confluence, and on the other hand there appears to be scope for trivial parallel implementations.

An interaction net system is specified by a set $\Sigma$ of agents, and a set $\mathcal{R}$ of interaction rules. One can think of agents as logical symbols (connectives) and the interaction rules specify the cut-elimination procedure. It is in this sense that interaction nets are a

generalisation of proof nets where the user defines the connectives of the logic. However, interaction nets are not about defining new logics, they are about defining rewrite systems analogous to term rewriting systems. Over the last years these rewriting systems have been used to define various systems: implementation models, algorithm animation, visual programming, simulating other rewriting systems, and even to model cut-elimination of linear logic [42, 44]. In the next paragraphs we recall some key results of interaction nets.

Interaction nets are particular kinds of graph rewriting systems which have constraints over both the construction of graphs and the corresponding rewrite rules. In fact they are so constrained that it is surprising how they capture all computable functions—it is possible to simulate a Turing Machine, so they are Turing complete. Unlike models such as Turing machines, cellular automata, $\lambda$-calculus or combinators, an interaction net computational step can be defined as a constant time operation, and the model allows for parallelism (many steps can take place at the same time). The model therefore is an interesting one if we are interested in cost models of computation, and also take advantage of possible parallelism.

The fact that they lend themselves to modelling efficient computation, for instance $\beta$-optimal and efficient reduction (see for instance [8, 46]) is evidence to the fact that they can play an important rôle in computer science. There have been implementations of interaction nets since Lafont introduced them in 1990, and there are both textual and graphical representations of nets [58, 3]. In [45] an investigation began into the development of a programming language for interaction nets. This language has developed significantly over the last few years, and we contribute to it in this thesis.

We summarise some of the work done in interaction nets, grouping them into two areas.

### 1.2.1 Interaction nets as an implementation language

One of the earliest applications of interaction nets, and perhaps one of the biggest successes to date, is the encoding of the $\lambda$-calculus. The $\lambda$-calculus can be seen as a prototypical functional programming language, and moreover can be seen as the foundation for an implementation of a functional language. For several reasons however, the theory falls short of this ideal because the reduction steps are too "big" in that $\beta$-reduction $(\lambda x.t)u \to t[u/x]$ takes us out of the realms of the pure theory since substitutions, which are the hard part to implement, are not captured by the basic theory. Several possible solutions such as explicit substitutions and combinators have been put forward to solve this problem.

However, none of these solutions offer atomic computation steps and as a consequence they partially help to implement functional languages, but they do not offer any detail or insight about the cost of reduction, or help in understanding topics such as sharing for example.

The reduction steps in any interaction net on the other hand are always constant time operations by construction. This is one reason they have been very useful for implementing the $\lambda$-calculus. Further, sharing is very natural to capture in these nets, and in fact it is very difficult to duplicate computation. There have been a number of different encodings of the $\lambda$-calculus. We mention just a sample:

- Gonthier, Abadi and Lévy [24] gave an optimal implementation using an infinite set of (indexed) agents. In practice, this system turns out to be very inefficient in time, but several works (Asperti et al. [8] for example) have made significant performance improvements.

- In [41] an interaction system is given which uses a finite number of agents, but it does not implement substitution through $\lambda$-abstractions, which is essential to obtain sharing. Although very little sharing is captured by this system it does better than call-by-need. This system lead to a sequence of papers [43, 46] delivering more efficient evaluators. These systems are the most efficient to date.

- Lippi [40]: has given an alternative approach based on encoding the $\lambda$-calculus indirectly by implementing an environment machine.

Overall, these interaction net systems have given great insight into the $\lambda$-calculus, and also given the most efficient implementations of the $\lambda$-calculus to date.

Interaction nets have also been used to implement other languages: term rewriting systems [16], small functional languages, Prolog [13], etc. There is still more work to be done in this area, especially once a better understanding of parallelism for interaction nets can be achieved. We hope this thesis can assist in this respect.

### 1.2.2 Interaction nets as a programming language

Interaction nets were originally also put forward as a programming paradigm. Lafont demonstrated that interaction nets are a graphical programming language when they were introduced [36]. In particular, programming examples involving lists were given (for instance an append operation). Numbers were also used, using the constructors zero and successor. Other people, notably Lippi [39] investigated further the programming

paradigm, giving a sorting algorithm and an implementation of the Towers of Hanoi. Later, starting in [45], an approach to extend (see below) the visual aspects of interaction nets was developed (see for example [48, 47]). The motivation and the reasons for the success of writing algorithms and programs in interaction nets comes from the following points:

1. The graphical representation of the problem is directly cast into the graphical language. The algorithm given can be understood as programming directly with the internal data structures, rather than some syntax describing it. This therefore gives the programmer a more direct access to the structure of the problem. The programmer is able to be more efficient and we believe also that it is easier to write a correct program in the first instance.

2. All rewrite steps correspond to steps in the computation: there is no need to introduce additional data structures and operations that are not part of the problem.

3. Because of point 1 above, if we single-step the computation we get an animation of the algorithm directly from the rewriting system.

We will look at some example interaction net systems later. It was also useful, and necessary in some cases, to extend interaction nets to a richer programming language.

## 1.3 Implementing and extending interaction nets

In order to facilitate the use of interaction nets we need a framework to aid programming. Specifically, we need robust and efficient implementations and we need to extend interaction nets with rich programming constructs. The novelty of this thesis is that we develop the theory and practice of interaction nets at both the front-end (richer programming language constructs) and also the back-end (internal data-structures and low-level language).

There are several implementations of interaction nets in the public domain (for instance [58, 40, 38, 5]), which have been developed to demonstrate interaction nets graphically, or to test out various implementation ideas, such as parallelism. These are based on a "pure" calculus of interaction nets (see [17] for such a calculus and details of the operational model), and thus writing programs with these can be understood as analogous to writing functional programs with the pure $\lambda$-calculus. Although we can already program in interaction nets (they are after all Turing complete) they lack the structure that we

expect from modern programming languages. What was therefore missing was a substantial *programming language* development: syntax, semantics, implementation, as well as a development environment (tools to facilitate program development, such as graphical previewers).

In Chapter 3 we give in the background some detail of implementations, so we will not repeat that here. The first documented implementations were given in [20, 58, 39]. Some additional features were included in these implementations, and the work [45] started a richer approach which lifts interaction nets from the "pure" world to allow them to be used in practice. To give some analogies as what is being done here, consider the relationship between the $\lambda$-calculus and functional programming languages such as Haskell [55], Standard ML [54], or the relationship between Horn clauses and logic programming languages, such as Prolog, or finally, the relationship between the $\pi$-calculus and the language PICT [56].

In the list above, the programming language is there to provide not only some syntactical sugar, but also to provide features that the theory does not offer. For instance, if we look in detail at the analogy with the $\lambda$-calculus and functional programming languages, functional languages allow the definition of functions such as: `twice f x = f(f x)`, which is a significant improvement over $\lambda fx.f(fx)$ as a programming language, as programs can be reused for instance. With respect to interaction nets, Hassan et al. [29] provided a corresponding programming language called INET, which provides built-in constants such as integer and float numbers, Boolean values and strings, operations for those constants, conditional interaction rules, module system, input/output and side effects computation, and this has been built in Hassan's thesis [26]. Other extensions have been developed: multiple principal ports [2] and macros [59].

To facilitate some of the extensions, the theory of interaction nets has had to be developed. Features such as type systems [14], strategies of reduction [17], operational equivalence [18] and a semantics of interaction nets [52] have been developed. Finally, there are also parallel implementations [57, 34], where both MPI (Message Passing Interface) and GPU (Graphics Processing Units) have been used.

## 1.4 Contribution

This thesis builds on the research effort of using interaction nets to study computation as outlined above. Our main interests are the following topics, and we briefly explain the contributions made in each one.

- Standard implementation model: we give a standard implementation model written in the C language. There are a number of evaluators for interaction nets, and this model offers a unified implementation model for different encoding methods. It is useful to compare properties such as efficiency, and locality which is one of the important properties to exploit parallel computation in interaction nets. From this study, we also propose a method of encoding that has good locality properties.

- Textual calculus: we give a new textual calculus for interaction nets, which is close to the implementation model, and thus rewritings in this calculus correspond directly to the operations on the data-structures. We also show some properties of the calculus, such as correctness.

- Low-level language: we give a low-level language for interaction nets. This is not only close to the implementation models, but also offers a bytecode execution model. We believe that this language can help pave the way for a better theory of compilation of interaction nets, and in particular, we mention that we can look for more efficient implementation, for example reuse of memory in the compilation of rules (we mention this in some detail in future work).

- Extending interaction nets: in order to use interaction nets as a programming language, we add built-in constants such as integer numbers, tuples and lists, together with operations for those constants. In addition, we introduce conditional interaction rules and deeper pattern matching.

- Parallelism: we give a multi-threaded execution model of interaction nets that is suitable to execute on recent multi-core processors. This execution model is an instance of the low-level language, and the parallelism is naturally derived from the locality of the implementation model.

Our work is not the first attempt at addressing these topics: a number of abstract machines were developed in the early days of interaction nets (a number unpublished). We build on those ideas, after experience and hindsight, to build more efficient data structures. Our philosophy is to keep things very simple: we have one of the simplest data structures for representing nets for instance. Experimental evidence shown in this thesis demonstrates that this gives improvements on the current state of the art, and in addition it also leads us to the possibility of using this kind of technology more widely.

## 1.5 Thesis overview

The structure of the rest of the thesis, and the main contributions of this thesis are as follows.

**Chapter 2.** Background. Most of this is standard material, but we offer some original contributions by adding some novel examples. For example, we give an implementation of Gödel's System T as one of the examples that is the topic of a paper presented at the 5th International Workshop on Graph Computation Models (GCM 2014) [50]. Some additional examples given in this chapter are used for producing benchmarks in later chapters where we compare different evaluators.

**Chapter 3.** Here we give some details about related work by examining other implementations of interaction nets developed to date. These evaluators include PIN, INET and amineLight, published in the proceeding of the 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008) [28], the Fifth International Workshop on Computing with Terms and Graphs (TERM-GRAPH 2009) [29] and 9th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2010) [30] respectively. In particular, we focus on the internal data structures of these implementations and give the standard implementation model. Finally we discuss the relative merits of each one.

**Chapter 4.** In this chapter we give the details of our proposed implementation based on a single link encoding of the internal data-structure. This is simpler, but has indirection nodes. In addition to the definition of the data-structures and rewrite rules, this chapter also contains a contribution in the form of a textual calculus based on this data-structure. We prove correctness and show various properties. Novel feature: rules correspond directly to the data-structure (i.e. the calculus is exactly the same as the data-structure so we can reason about it in a textual way). (Lightweight calculus was published in the proceedings of GT-VMT 2010 [30]).

**Chapter 5.** In this chapter we introduce our low-level language for interaction nets. We give instructions to manipulate the data-structure, and also show how to compile the calculus to a list of instructions. We also provide lots of examples and details so that this work is easily re-produced by anyone else wanting to contribute to this area. Moreover, we give a correspondence to the standard implementation model and show a bytecode execution model to interpret those instructions.

The previous two chapters: the calculus and the low-level language were presented at TERMGRAPH 2014 [31].

**Chapter 6.** Extending interaction nets. A main contribution is pattern matching, which was published in the proceedings of the Fourth International Workshop on Computing with Terms and Graphs (TERMGRAPH 2007) [32] (implementation issues were published in the proceedings of the Tenth International Workshop on Rule-Based Programming (RULE 2009) [27]), and we illustrate this mechanism with examples. We also give an extension of our low-level language and compilation to cover this extension.

**Chapter 7.** We implement a bytecode interpreter, which is an evaluator of the low-level language, and we also introduce a multi-threaded execution model. We compare the performance with other evaluators, and thus in this chapter the benchmark results demonstrate the usefulness of the techniques developed. In addition to the results, we give in this chapter the further work that we have identified. In particular, we focus on parallel issues that have been in view for all the other works presented in the thesis, and we demonstrate with benchmark results that have recently presented at DCM 2014 [51].

**Chapter 8.** Finally we conclude the thesis by providing a commentary on the work done.

# Chapter 2

# Background

In this chapter we review the basic notions of interaction nets [36]. First, we recall the original graphical presentation given by Lafont [36] then we review a textual calculus for interaction nets presented by Fernández and Mackie [17].

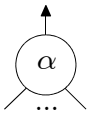
We include a number of examples that demonstrate the usefulness of interaction nets and also show how they are used in different ways. In particular, we show an implementation of Gödel's System T (an original contribution presented at the 5th International Workshop on Graph Computation Models—GCM 2014) in addition to some standard examples such as the Fibonacci and Ackermann functions that are very useful for generating a large number of interactions in the benchmark results that we shall give later in the thesis.

## 2.1 Interaction nets

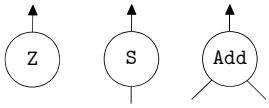
In this section we recall the graph rewriting formalism of interaction nets. In Section 2.1.1 we review the original presentation proposed by Lafont [36], and in Section 2.1.2 we review a textual calculus of interaction nets [17]. Both graphical and textual presentations are equal, but each formalism has its own advantages: the textual calculus is better for proving some properties and is compact to write down, whereas the graphical presentation gives a visual and hence clearer presentation of the rules.

### 2.1.1 Graph rewriting system

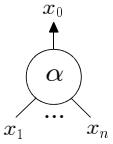
A *net* is an undirected graph with labelled vertices called *agents*. An agent with a *symbol* $\alpha$ as a label is called an agent $\alpha$. Each agent $\alpha$ has one *principal port*, depicted by an arrow, and (fixed) $n$ *auxiliary ports* as follows:



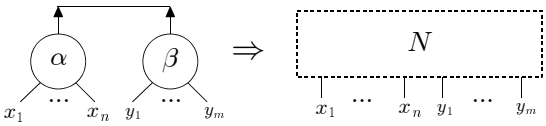

The number of auxiliary ports is called the *arity*, and each agent $\alpha$ has an associated arity $n$ given by a function $ar(\alpha) = n$. The following is an example of agents `Z`, `S`, `Add` with arity $0, 1, 2$ respectively for unary natural numbers and the addition operation:




Each port is connected with at most one port of an agent. A port which is not connected with another port is called *free*. A set of free ports is called an *interface*. We may use labels for an interface to distinguish these ports as follows:




A pair of agents which are connected together by their principal ports is called an *active pair*. A rewriting of a net is performed only on an active pair according to an *interaction rule*. At most one interaction rule exists for each active pair, and the interface is preserved before and after the rewriting as shown in the next figure, where $N$ is any net which contains no active pairs:




We may write this rule as $(\alpha, \beta) \Rightarrow N$.

Interaction nets is a graph rewriting system, which is specified by a pair consisting of a symbol set $\Sigma$ and an interaction rule set $\mathcal{R}$. We use $\Sigma$ to range over symbol sets and $\mathcal{R}$ to range over rule sets. For instance, in the above example of agents for natural numbers and the addition operation, the symbol set is {`Z`, `S`, `Add`} and the rule set for addition is given in Figure 2.1. By using these rules, a net representing $1 + 0$ is reduced to a net representing 1, as shown in the example reduction in Figure 2.1.

We write $N_1 \to N_2$ when $N_1$ reduces in one step to $N_2$. Interaction nets have the following property because there is at most one interaction rule for each active pair [36]:

**Theorem 2.1.1 (Strong confluence (diamond property))**
*Let $N$ be a net. If $N \to N_1$ and $N \to N_2$ with $N_1 \neq N_2$, then there is a net $N_3$ such that $N_1 \to N_3$ and $N_2 \to N_3$.* □

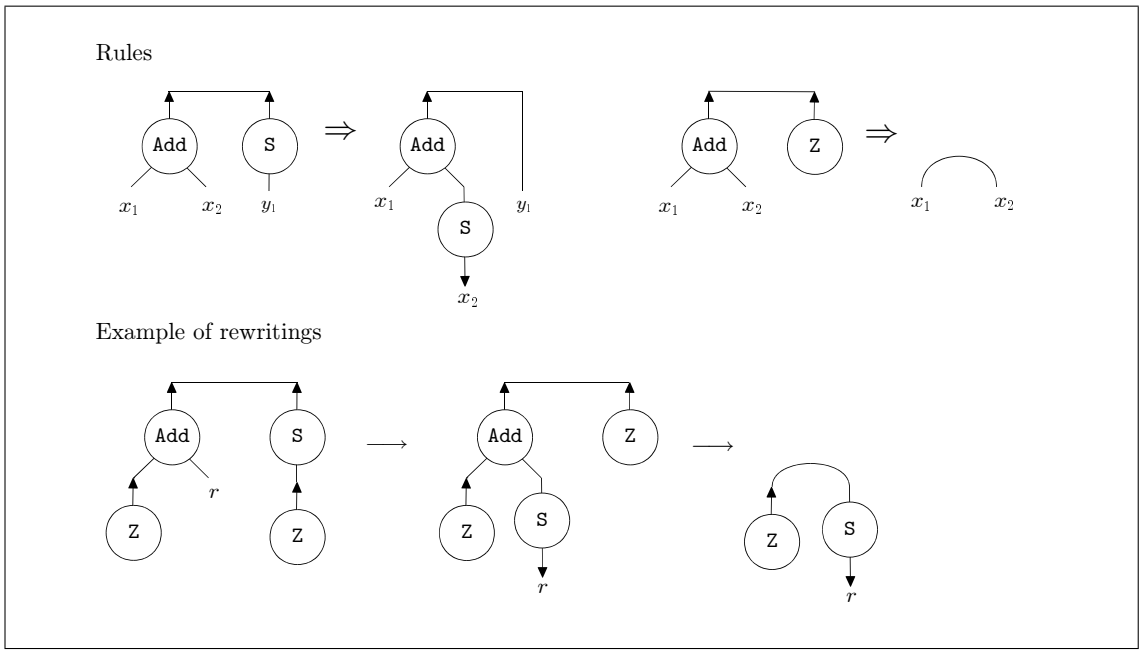



Figure 2.1: An example of rules and rewritings of interaction nets

$N_1$ is called a *normal form* when there is no $N_2$ such that $N_1 \to N_2$. We write $N \Downarrow N_1$ when $N \to^* N_1$ and $N_1$ is a normal form. By strong confluence, the following holds:

**Theorem 2.1.2 (Determinacy)**
*If $N \Downarrow N_1$ and $N \Downarrow N_2$, then $N_1 = N_2$.* □

*Locality* is a property of rewriting such that there is at most one interaction rule for each active pair and the interface is preserved during the rewriting. By the locality property, all rewritings are performed locally. In interaction nets, since strong confluence holds and all rewrites are local, rewriting can be performed in any order. Therefore interaction nets are inherently parallel.

### 2.1.2 A textual calculus for interaction nets

In this section, we review the textual calculus proposed by Fernández and Mackie [17]. This can be considered as a theoretical development of a syntactical notation described by Lafont [36], extended with a rewriting mechanism.

First, we review definitions of terms and equations for representing nets:

**Agents:** Let $\Sigma$ be a set of symbols, ranged over by $\alpha, \beta, \ldots$, each with a given *arity* $\mathsf{ar} : \Sigma \to \mathbb{N}$. An occurrence of a symbol will be called an *agent*. The arity of a symbol corresponds precisely to the number of auxiliary ports.

**Names:** Let $\mathcal{N}$ be a set of names, ranged over by $x, y, z$, etc. $\mathcal{N}$ and $\Sigma$ are assumed disjoint. Names correspond to wires in the graphical system.

**Terms:** A *term* is built on $\Sigma$ and $\mathcal{N}$ by the grammar: $t ::= x \mid \alpha(t_1, \ldots, t_n)$, where $x \in \mathcal{N}$, $\alpha \in \Sigma$, $\mathsf{ar}(\alpha) = n$ and $t_1, \ldots, t_n$ are terms, with the restriction that each name can appear at most twice. If $n = 0$, then we omit the parentheses. If a name occurs twice in a term, we say that it is *bound*, otherwise (i.e. occurs once) it is *free*. We write $s, t, u$ to range over terms, and $\vec{s}, \vec{t}, \vec{u}$ to range over sequences of terms. We use $\mathcal{T}$ as the set of terms. A term of the form $\alpha(t_1, \ldots, t_n)$ can be seen as a tree with the principal port of $\alpha$ at the root, and where the terms $t_1, \ldots, t_n$ are the subtrees connected to the auxiliary ports of $\alpha$.

**Equations:** If $t$ and $u$ are terms, then the pair $t = u$ is an *equation*. $\Delta$, $\Theta, \ldots$ will be used to range over multisets of equations. An occurrence of an equation corresponds to a connection between two ports.

**Configurations:** A *configuration* is a pair: $\langle\ \vec{t} \mid \Delta\ \rangle$, where $\vec{t}$ is a sequence $t_1, \ldots, t_n$ of terms, and $\Delta$ a multiset of equations. Each variable occurs at most twice in a configuration. Configurations that differ only on names are considered equivalent. A name that occurs exactly once is *free*, and a name that occurs twice is *bound*. We use $C, C'$ to range over configurations. We call $\vec{t}$ the *head* and $\Delta$ the *body* of a configuration.

In this notation, we can obtain a configuration from a net by using the following translation [49]:

- **Agents:** For every agent $\alpha$, we introduce a term $\alpha(x_1, \ldots, x_n)$ where each of $x_1, \ldots, x_n$ is a fresh name. The occurrence of the term $\alpha(x_1, \ldots, x_n)$ in this translation corresponds to the principal port of the agent $\alpha$, and each occurrence of $x_1, \ldots, x_n$ corresponds to the free auxiliary ports respectively.

- **Connections between two principal ports:** We assume that terms for these principal ports are $\alpha(\vec{t})$ and $\beta(\vec{s})$. For this connection, we introduce an equation $\alpha(\vec{t}) = \beta(\vec{s})$.

- **Connections between a principal port and a free auxiliary port:** We assume that terms for a principal port and an auxiliary port are $\alpha(\vec{t})$ and $x$ respectively. For this connection, we replace the occurrence of $x$ with $\alpha(\vec{t})$.

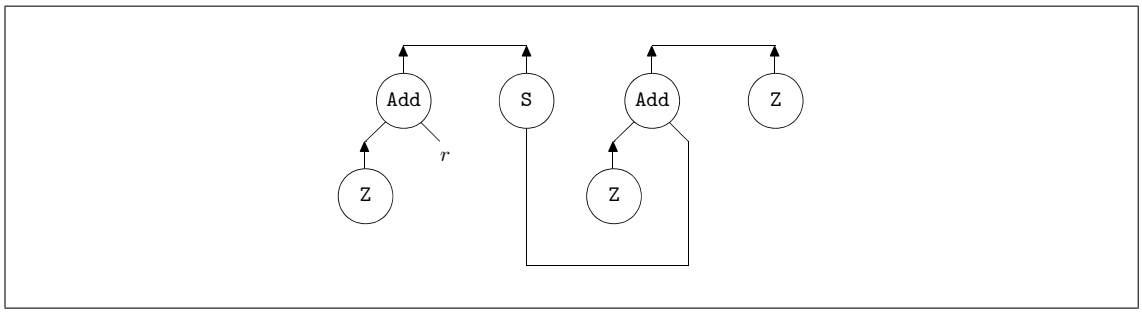
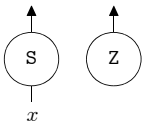



Figure 2.2: An example of nets

- **Connections between two free auxiliary ports:** We assume that terms for two free auxiliary ports are $x$ and $y$ respectively. For this connection, we introduce a fresh name $z$ and we replace the occurrence of $x$ and $y$ with $z$.

**Example 2.1.3**

*Let us consider the nets in Figure 2.1. First, we consider the sub-net on the right. For the following agent* S *with a labelled auxiliary port* $x$ *and agent* Z*,*




*we obtain terms* $\mathtt{S}(x)$ *and* Z*, and for the result of connecting the agent* Z *to the auxiliary port of the agent* S*, we obtain a term* $\mathtt{S}(\mathtt{Z})$*. Then, for the net, we obtain the following configuration by collecting* $\mathtt{S}(\mathtt{Z})$ *into an interface:*

$$\langle\ \mathtt{S}(\mathtt{Z})\ \mid\ \rangle.$$

*For the net on the most left-hand side in the figure, we obtain the following configuration because a connection between two principal ports is represented as an equation:*

$$\langle\ r\ \mid\ \mathtt{Add}(\mathtt{Z},\ r) = \mathtt{S}(\mathtt{Z})\ \rangle.$$

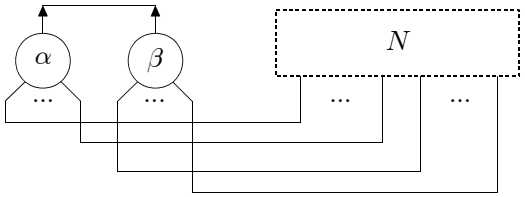
**Example 2.1.4**

*As another example, let us consider the net in Figure 2.2 which has a connection between two auxiliary ports. Because agents* S *and* Add *are connected by their auxiliary ports, we obtain the following configuration by introducing a fresh name* $w$ *corresponding to the auxiliary ports in* S *and* Add*:*

$$\langle\ r\ \mid\ \mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(w),\ \mathtt{Add}(\mathtt{Z}, w) = \mathtt{Z}\ \rangle.$$

Next, we review the syntax of rewriting rules. In interaction nets, a rewriting is performed only on an active pair (in a given net) according to an interaction rule, and

each auxiliary port of the active pair is preserved before and after the rewriting. Therefore it is essential to know how the auxiliary ports of a rewrite rule are connected. For this purpose, Lafont [36] proposed a compact notation by connecting auxiliary ports between the left-hand side net and the right hand side net of an interaction rule:

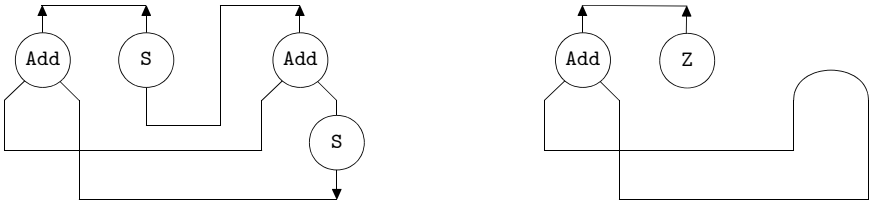



We can now represent a rule as an equation $\alpha(t_1, \ldots, t_n) = \beta(s_1, \ldots, s_k)$. In order to identify the equation as an interaction rule, we use $\bowtie$ instead of $=$ as follows:

$$\alpha(t_1, \ldots, t_n) \bowtie \beta(s_1, \ldots, s_k).$$

**Example 2.1.5**

*The interaction rules for addition on the natural numbers in Figure 2.1 can be described using the nets:*




*and these are represented syntactically using the equations:*

$$\mathtt{Add}(y, \mathtt{S}(w)) \bowtie \mathtt{S}(\mathtt{Add}(y, w)), \qquad \mathtt{Add}(y, y) \bowtie \mathtt{Z}$$

To summarise *interaction rules* have the form:

$$\alpha(t_1, \ldots, t_n) \bowtie \beta(s_1, \ldots, s_k)$$

where $\alpha(t_1, \ldots, t_n)$ and $\beta(s_1, \ldots, s_k)$ are terms. All names occur exactly twice in a rule, and there should be at most one rule between $\alpha$ and $\beta$ in the set of rules $\mathcal{R}$ of an interaction nets system. $\mathcal{R}$ is closed under symmetry, thus if $\alpha(\vec{t}) \bowtie \beta(\vec{s}) \in \mathcal{R}$ then $\beta(\vec{s}) \bowtie \alpha(\vec{t}) \in \mathcal{R}$.

**Definition 2.1.6 (Names in terms)**

*The set* $\mathsf{Name}(t)$ *of names of a term* $t$ *is defined in the following way, which extends to sequences of terms, equations, sequences and multisets of equations, and rules in the obvious way.*

$$\begin{aligned} \mathsf{Name}(x) &= \{x\}, \\ \mathsf{Name}(\alpha(t_1, \ldots, t_n)) &= \mathsf{Name}(t_1) \cup \cdots \cup \mathsf{Name}(t_n). \end{aligned}$$

**Definition 2.1.7 (Linear)**

*When every name occurs twice in a term $t$, then we say $t$ is* linear. *We extend this notion into equations, sequences of terms, multisets of equations, and configurations.*

We can replace its free names by new names, provided the linearity restriction is preserved.

**Definition 2.1.8 (Substitution)**

*The notation $t[u/x]$ denotes a substitution that replaces the free occurrence of $x$ by the term $u$ in $t$. We only consider substitutions that preserve the linearity of the terms. Remark that since the name $x$ occurs exactly once in the term, this operation can be implemented directly as an assignment, as is standard in the linear case. This notion extends to sequence of terms, equations, sequences and multisets of equations, and configurations in the obvious way.*

The reduction rules in the calculus are divided into three kinds: *Indirection* which binds fragments of connections, *Collect* which records the result of computation into the interface, and *Interaction* which performs the actual computation according to interaction rules:

**Indirection:**

$\langle\ \vec{t}\ \mid\ x = t, u = s, \Delta\ \rangle \longrightarrow \langle\ \vec{t}\ \mid\ u[t/x] = s, \Delta\ \rangle$ where $x$ occurs in $u$.

**Collect:**

$\langle\ \vec{t}\ \mid\ x = s, \Delta\ \rangle \longrightarrow \langle\ \vec{t}[s/x]\ \mid\ \Delta\ \rangle$ where $x$ occurs in $\vec{t}$.

**Interaction:**

$\langle\ \vec{t}\ \mid\ \alpha(\vec{t_1}) = \beta(\vec{t_2}), \Delta\ \rangle \longrightarrow \langle\ \vec{t}\ \mid\ \vec{t_1} = \vec{s^r}, \vec{t_2} = \vec{u^r}, \Delta\ \rangle$

where $\alpha(\vec{s}) \bowtie \beta(\vec{u}) \in \mathcal{R}$ and $\vec{s^r}$ and $\vec{u^r}$ are the result of replacing each occurrence of a name $x$ for $\alpha(\vec{s}) \bowtie \beta(\vec{u})$ by a fresh name $x^r$ respectively.

**Example 2.1.9**

*The following is a possible reduction sequence for the most left-hand side net in Figure 2.1:*

$\langle\ r\ \mid\ \mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(\mathtt{Z})\ \rangle$

$\longrightarrow\ \langle\ r\ \mid\ \mathtt{Z} = y', r = \mathtt{S}(w'), \mathtt{Z} = \mathtt{Add}(y', w')\ \rangle$ (Interaction)

$\longrightarrow\ \langle\ r\ \mid\ r = \mathtt{S}(w'), \mathtt{Z} = \mathtt{Add}(\mathtt{Z}, w')\ \rangle$ (Indirection)

$\longrightarrow\ \langle\ \mathtt{S}(w')\ \mid\ \mathtt{Z} = \mathtt{Add}(\mathtt{Z}, w')\ \rangle$ (Collect)

$\longrightarrow\ \langle\ \mathtt{S}(w')\ \mid\ \mathtt{Z} = w'', w' = w''\ \rangle$ (Interaction)

$\longrightarrow\ \langle\ \mathtt{S}(w')\ \mid\ w' = \mathtt{Z}\ \rangle$ (Indirection)

$\longrightarrow\ \langle\ \mathtt{S}(\mathtt{Z})\ \mid\ \rangle.$ (Collect)

During rewritings, linearity is preserved, and so the following holds:

**Theorem 2.1.10 (Linearity)**
*Let $C$ be a linear configuration. If $C \longrightarrow C_1$, then $C_1$ is also linear.* □

We define $C_1 \Downarrow C_2$ by $C_1 \rightarrow^* C_2$ where $C_2$ is in normal form. In contrast with this normal form, as in the $\lambda$-calculus, there are two types of normal form: full normal form and weak normal form. In our framework, we can define *interface normal form (INF)* as a weak normal form paying attention to interfaces [17]:

**Definition 2.1.11 (Interface normal form (INF))**
*A configuration $\langle\ \vec{t} \mid \Delta\ \rangle$ is in* interface normal form (INF) *when every $t_i$ in the interface $\vec{t}$ has one of the following forms:*

- *an agent $\alpha(\vec{s})$,*
- *a name $x$ that occurs in $\vec{t}$ except for the $t_i$,*
- *a name $x$ that occurs in an irreducible equation in $\Delta$.*

Intuitively, a configuration is in INF when it is not expected to obtain new results that could be observed from the interface even if some reductions were applied.

To obtain a normal form in INF, we define a reduction strategy called *weak reduction* [17]:

**Definition 2.1.12 (Weak reduction)**
*For a given configuration $\langle\ t_1, \ldots, x, \ldots, t_n \mid s = u, \Delta\ \rangle$, we apply any rule only to $s = u$ in which $x$ occurs.*

By using weak reduction, we can evaluate only active pairs that are connected to the interface. In other words, we avoid evaluation of nets which are disconnected from the interface. This reduction strategy will be particularly useful when we have infinite lists, encodings of recursive functions, etc.

## 2.2 Examples

In this section we give some examples of interaction nets. First, we show two examples involving arithmetic operations: a system of interaction nets to compute Fibonacci numbers and another to compute the Ackermann function. These examples demonstrate how mathematical functions can be implemented in interaction nets. Next, we show a more elaborate example, giving an encoding of Gödel's System $\mathcal{T}$. Some parts of this system

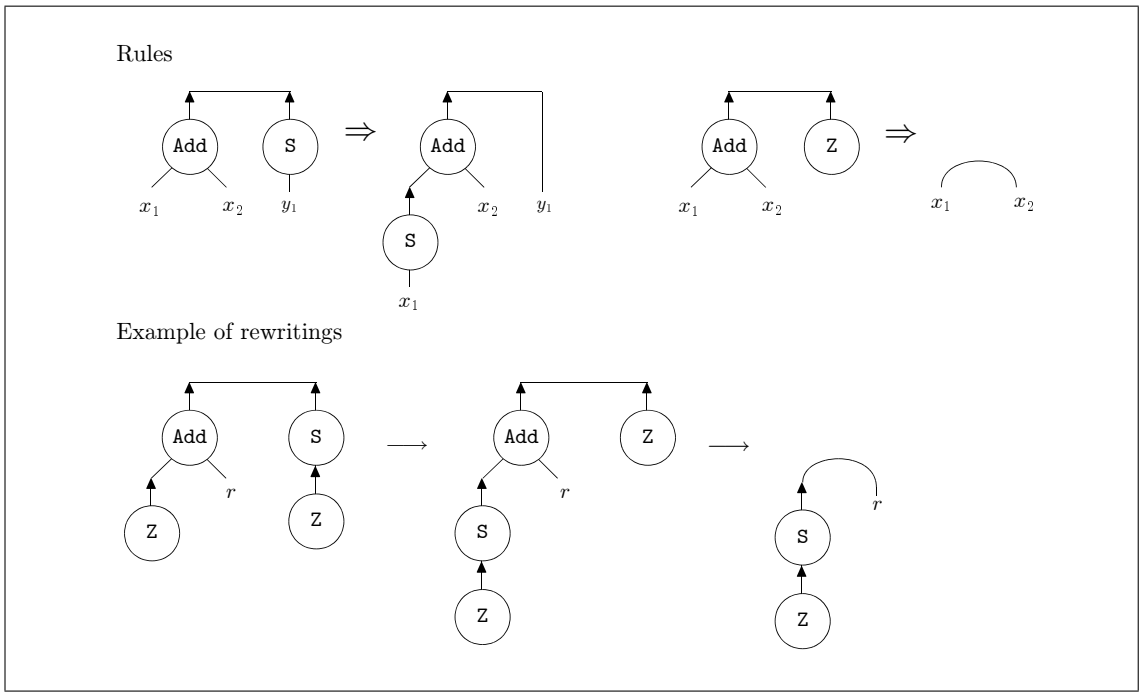
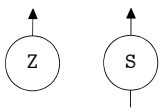



Figure 2.3: Alternative rules of the addition on unary natural numbers

are a simplification, or refinement, of some of the encodings of the $\lambda$-calculus, for example the YALE encoding [43].

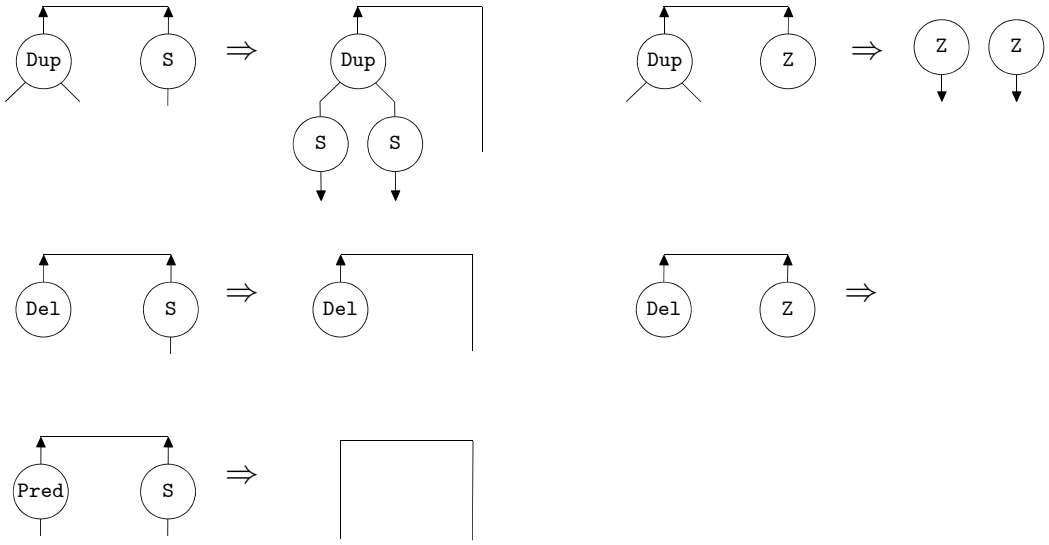
### 2.2.1 Arithmetic operations on unary natural numbers

Unary natural numbers are built by the zero `0` and the successor $\mathtt{S}(x)$ for a natural number $x$. In interaction nets this is expressed by using the following two agents `Z` and `S` as shown in Section 2.1.1:




The addition operation is defined in Figure 2.1. Here, we introduce alternative rules as shown in Figure 2.3, and the result of the addition of $\mathtt{add}(n, m)$ for natural numbers $n$ and $m$ is obtained as the computation result of $\mathtt{Add}(\bar{m}, r) = \bar{n}$, where $\bar{m}$ and $\bar{n}$ are nets of unary natural numbers for $m$ and $n$. The difference is discussed in Section 7.2.2.

We define basic operations for duplication, erasing and predecessor:




These rules are written textually as follows:

$$
\begin{array}{lcl}
\texttt{Add}(x_1, x_2) = \texttt{Z} & \Rightarrow & x_1 = x_2 \\
\texttt{Add}(x_1, x_2) = \texttt{S}(y_1) & \Rightarrow & \texttt{Add}(\texttt{S}(x_1), x_2) = y \\
\texttt{Dup}(a_1, a_2) = \texttt{Z} & \Rightarrow & a_1 = \texttt{Z}, a_2 = \texttt{Z} \\
\texttt{Dup}(a_1, a_2) = \texttt{S}(x) & \Rightarrow & a_1 = \texttt{S}(w_1), a_2 = \texttt{S}(w_2), x = \texttt{Dup}(w_1, w_2) \\
\texttt{Del} = \texttt{Z} & \Rightarrow & \\
\texttt{Del} = \texttt{S}(x) & \Rightarrow & \texttt{Del} = x \\
\texttt{Pred}(a) = \texttt{S}(x) & \Rightarrow & a = x
\end{array}
$$

We use those rules in the following Fibonacci number and Ackermann function.

**Fibonacci number** The definition of Fibonacci number $F_n$ is as follows:

$$
\begin{cases}
F_0 &= 1 \\
F_1 &= 1 \\
F_n &= F_{n-2} + F_{n-1}
\end{cases}
$$

The Fibonacci function can be encoded using the interaction net system given in Figure 2.4.

These rules are written textually as follows:

$$
\begin{array}{lcl}
\texttt{Fib}(r) = \texttt{Z} & \Rightarrow & r = \texttt{S}(\texttt{Z}) \\
\texttt{Fib}(r) = \texttt{S}(x) & \Rightarrow & \texttt{Fib2}(r) = x \\
\texttt{Fib2}(r) = \texttt{Z} & \Rightarrow & r = \texttt{S}(\texttt{Z}) \\
\texttt{Fib2}(r) = \texttt{S}(x) & \Rightarrow & x = \texttt{Dup}(x_1, x_2), \\
 & & \texttt{Fib}(r_1) = \texttt{S}(x_1), \texttt{Fib}(r_2) = x_2, \texttt{Add}(r_2, r) = r_1
\end{array}
$$

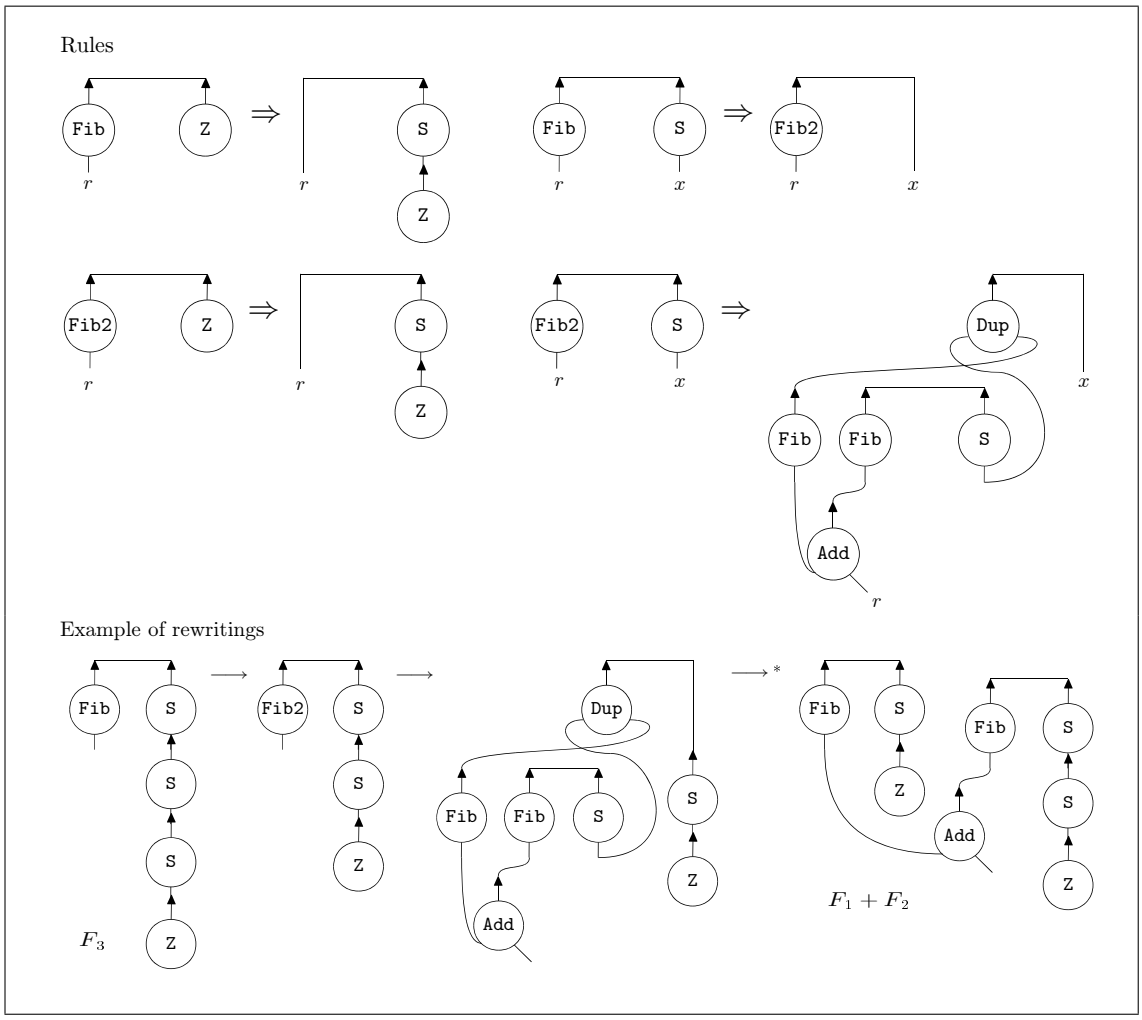



Figure 2.4: Fibonacci number on the unary natural number

The number $F_n$ is obtained by evaluation of $\texttt{Fib}(r) = \bar{n}$ where $\bar{n}$ is a unary natural number with `S` and `Z` of $n$. For instance, $F_3$ is obtained as the computation result of $\texttt{Fib}(r) = \texttt{S}(\texttt{S}(\texttt{S}(\texttt{Z})))$.

**Ackermann function** The definition of Ackermann function $A$ is as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } n = 0 \\ A(m-1, A(m, n-1)) & \text{otherwise} \end{cases}$$

The Ackermann function can be represented using the interaction net system given in Figure 2.5. These rules in this figure are written textually as follows:

$$\begin{aligned} \texttt{A}(y,r) = \texttt{Z} \quad &\Rightarrow \quad r = \texttt{S}(y) \\ \texttt{A}(y,r) = \texttt{S}(x) \quad &\Rightarrow \quad \texttt{A2}(\texttt{S}(x), r) = y \\ \texttt{A2}(x,r) = \texttt{Z} \quad &\Rightarrow \quad \texttt{Pred}(\texttt{A}(\texttt{S}(\texttt{Z}), r)) = x \\ \texttt{A2}(x,r) = \texttt{S}(y) \quad &\Rightarrow \quad x = \texttt{Dup}(\texttt{Pred}(\texttt{A}(w,r)), \texttt{A}(y,w)) \end{aligned}$$

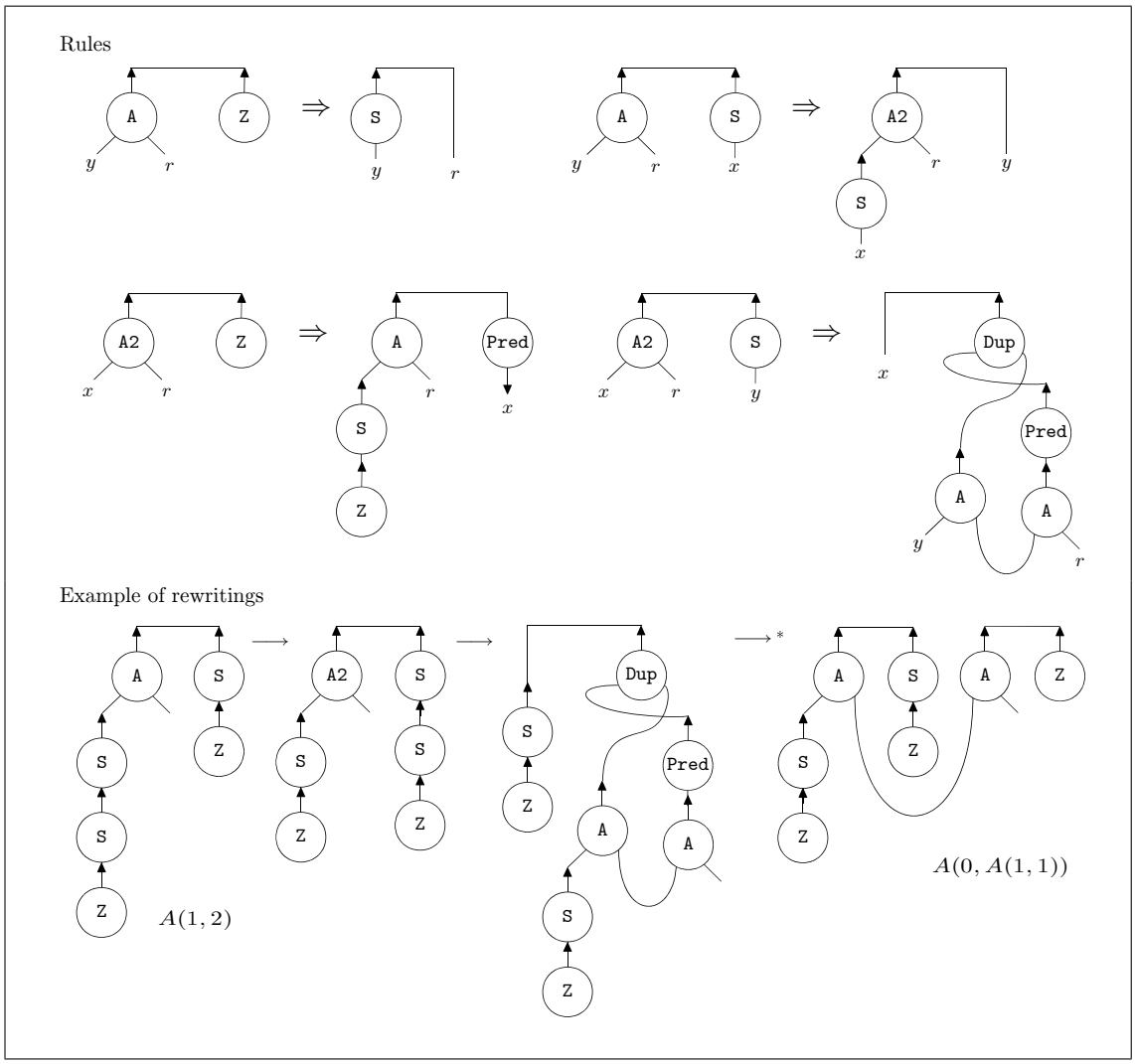



Figure 2.5: Ackermann function on the unary natural number

The computation result of $A(m, n)$ is obtained by evaluation of $\mathtt{A}(\bar{n}, r) = \bar{m}$ where $\bar{m}$ and $\bar{n}$ are unary natural numbers with `S` and `Z` of $m$ and $n$ respectively. For instance, $A(1, 3)$ is obtained as the computation result of $\mathtt{A}(\mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{Z}))), r) = \mathtt{S}(\mathtt{Z})$.

### 2.2.2 Gödel's System T

Gödel's System T [23] is the simply typed $\lambda$-calculus, with function and product types, extended with natural numbers. It is a very simple system, yet has enormous expressive power—well beyond that of primitive recursive functions. We show how to encode this system using interaction nets because it illustrates some of the ideas of encoding the $\lambda$-calculus, and it is also an original contribution. We will assume some knowledge of the $\lambda$-calculus and also of Gödel's System T.

Specifically, this example brings together on one hand the successful study of encoding $\lambda$-calculus and related systems into interaction nets, together with the result that Gödel's

| **Terms** | **Variable Constraint** | **Free Variables (fv)** |
|---|---|---|
| $x$ | – | $\{x\}$ |
| $tu$ | $\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$ | $\mathsf{fv}(t) \cup \mathsf{fv}(u)$ |
| $\lambda p.t$ | $\mathsf{bv}(p) \subseteq \mathsf{fv}(t)$ | $\mathsf{fv}(t) \setminus \mathsf{bv}(p)$ |
| $\langle p, q\rangle$ | $\mathsf{fv}(p) \cap \mathsf{fv}(q) = \varnothing$ | $\mathsf{fv}(p) \cup \mathsf{fv}(q)$ |
| $0$ | – | $\varnothing$ |
| $\mathsf{S}\ t$ | – | $\mathsf{fv}(t)$ |
| $\mathsf{iter}\ t\ u\ v$ | $\mathsf{fv}(t) \cap \mathsf{fv}(u) = \mathsf{fv}(u) \cap \mathsf{fv}(v) = \varnothing$<br>$\mathsf{fv}(t) \cap \mathsf{fv}(v) = \varnothing$ | $\mathsf{fv}(t) \cup \mathsf{fv}(u) \cup \mathsf{fv}(v)$ |
| **Pattern** | **Variable Constraint** | **Bound Variables (bv)** |
| $x$ | – | $\{x\}$ |
| $\langle p, q\rangle$ | $\mathsf{bv}(p) \cap \mathsf{bv}(q) = \varnothing$ | $\mathsf{bv}(p) \cup \mathsf{bv}(q)$ |

Table 2.1: Terms

System T can be encoded with the linear $\lambda$-calculus and an iterator [4]. What this latter result says is that there are redundancies in Gödel's System T—copying and erasing can be done either by the iterator or by the $\lambda$-calculus. We can remove the copy and erasing power of the $\lambda$-calculus, and still keep the expressive power. Table 2.1 summarises the syntax of our linear version of System T. The first four lines give the linear $\lambda$-calculus with pairs. The construct $\lambda p.t$ is the usual abstraction, extended to allow patterns of variables or pairs of patterns (as defined at the bottom of the table). The remaining three rules define the syntax for constructing numbers and the iteration. We work with terms modulo $\alpha$-conversion as usual. In Figure 2.6 we give the typing rules for the calculus. The syntax judgements are written as $p_1 : A_1, \ldots, p_n : A_n \vdash t : B$, and the typing rules capture the linear variable constraints.

The reduction rules for calculus are given below:

| Reduction | | | Condition |
|---|---|---|---|
| $(\lambda p.t)v$ | $\longrightarrow$ | $[p \ll v].t$ | $\mathsf{fv}(v) = \varnothing$ |
| $\mathsf{iter}\ (\mathsf{S}\ t)\ u\ v$ | $\longrightarrow$ | $\mathsf{iter}\ t\ (vu)\ v$ | $\mathsf{fv}(v) = \varnothing$ |
| $\mathsf{iter}\ 0\ u\ v$ | $\longrightarrow$ | $u$ | $\mathsf{fv}(v) = \varnothing$ |

The construct $[p \ll v].t$ is a matching operation, defined as:

$$
\begin{array}{lcl}
[x \ll v].t & \longrightarrow & t[v/x] \\
[\langle p, q\rangle \ll \langle t, u\rangle].t & \longrightarrow & [p \ll t].[q \ll u].t
\end{array}
$$

**Context**

$$\frac{}{x : A \vdash x : A}\ (\mathsf{Var}) \qquad \frac{\Gamma, p : A, q : B \vdash t : C}{\Gamma, \langle p, q\rangle : A \otimes B \vdash t : C}\ (\mathsf{Pattern\ Pair})$$

$$\frac{\Gamma, p : A, q : B, \Delta \vdash t : C}{\Gamma, q : B, p : A, \Delta \vdash t : C}\ (\mathsf{Exchange})$$

**Logical Rules**:

$$\frac{\Gamma, p : A \vdash t : B}{\Gamma \vdash \lambda p.t : A \multimap B}\ (\multimap\mathsf{Intro}) \qquad \frac{\Gamma \vdash t : A \multimap B \qquad \Gamma \vdash u : A}{\Gamma \vdash tu : B}\ (\multimap\mathsf{Elim})$$

$$\frac{\Gamma \vdash t : A \qquad \Delta \vdash u : B}{\Gamma, \Delta \vdash \langle t, u\rangle : A \otimes B}\ (\mathsf{Pair})$$

**Numbers:**

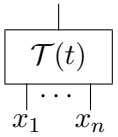
$$\frac{}{\Gamma \vdash 0 : \mathsf{nat}}\ (\mathsf{Zero}) \qquad \frac{\Gamma \vdash t : \mathsf{nat}}{\Gamma \vdash \mathsf{S}\ t : \mathsf{nat}}\ (\mathsf{Succ})$$

$$\frac{\Gamma \vdash t : \mathsf{nat} \quad \Delta \vdash u : A \quad \Theta \vdash v : A \multimap A}{\Gamma, \Delta, \Theta \vdash \mathsf{iter}\ t\ u\ v : A}\ (\mathsf{Iter})$$

Figure 2.6: Linear System T

Substitution is a meta-operation defined as usual, and reductions can take place in any context. Matching forces evaluation of terms, and will always succeed. The conditions on the rules are used to preserve the linearity of those terms.

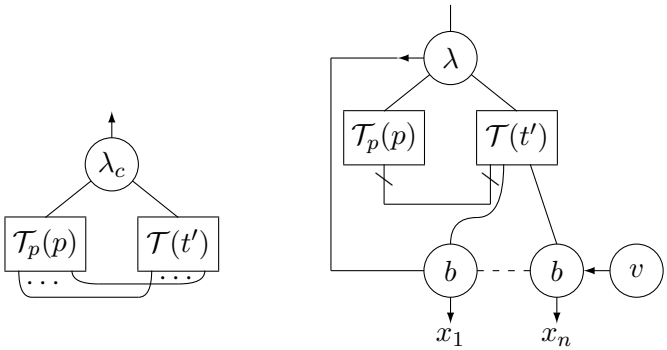
We can now give a translation $\mathcal{T}(\cdot)$ of linear System T terms into interaction nets. A term $t$ with $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$ is translated as a net $\mathcal{T}(t)$ with the root edge at the top, and $n$ free edges corresponding to the free variables:



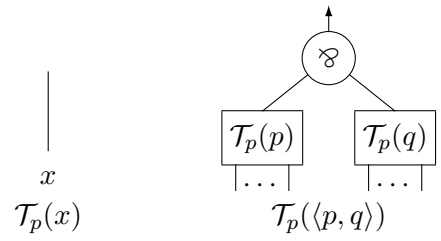

The agents needed for this compilation will be introduced, and then we give the interaction rules at the end.

**Variable.** When $t$ is a variable, say $x$, then $\mathcal{T}(t)$ is translated into an edge:


**Abstraction.** If $t$ is an abstraction, say $\lambda p.t'$, then there are two alternative translations of the abstraction, which are given as follows. The one of the left corresponds to a closed abstraction (when there are no free variables), and the one of the right has free variables $x_1 \cdots x_n$.

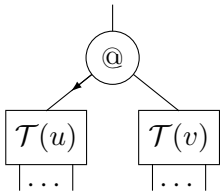



In these diagrams, we use an auxiliary function for the translation of patterns $\mathcal{T}_p(p)$ which is given by the following two rules.

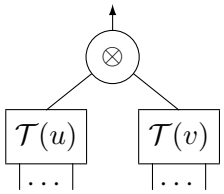



If $p$ is a variable, then it is translated into an edge. Otherwise, if it is a pair pattern, then it is translated as shown in the right-hand diagram above.

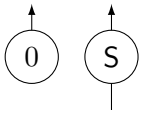
**Application.** If $t$ is an application, say $uv$, then $\mathcal{T}(uv)$ is given by the following net, where we have introduced an agent @ of arity 2 corresponding to an application.



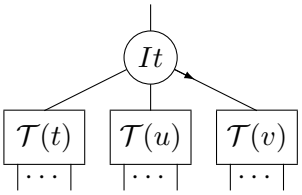

**Pair.** If $t$ is a pair, say $\langle u, v\rangle$, then $\mathcal{T}(\langle u, v\rangle)$ is given by the following net, where we have introduced an agent $\otimes$ of arity 2 corresponding to a pair.




**Numbers.** A number will be represented by a chain of successor agents (S), terminating with a zero (0) agent. S has one auxiliary port, and 0 has none:




**Iterator.** To encode iter $t$ $u$ $v$ we introduce one new agent as shown below. The principal port of this agent points to the function $v$, because we must wait for this to become a closed term before starting the interaction process.




In Figure 2.7 we give the net corresponding to the Ackermann function:

$$\mathcal{T}(\lambda m.(\mathsf{iter}\ m\ (\lambda x.\mathsf{S}\ x)\ (\lambda xy.\mathsf{iter}\ (\mathsf{S}\ y)\ (\mathsf{S}\ 0)\ x)))$$

(We have used $\eta$-conversion to slightly simplify this net.)

To complete this example, we give in Figure 2.8 most of the interaction rules for this system. The first rule deals with $\beta$-reduction, the next with pair pattern matching, and

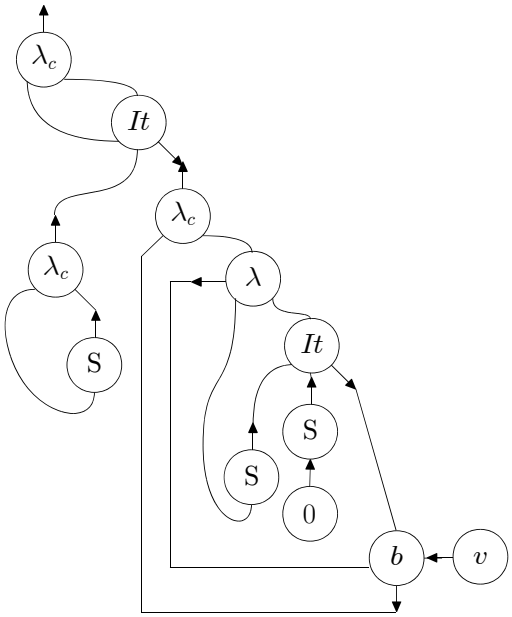
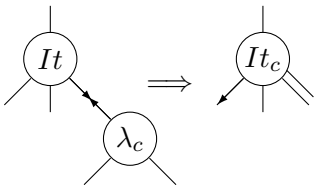


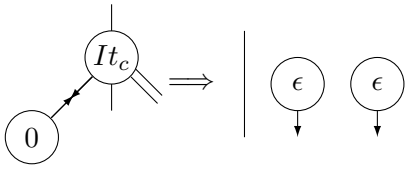

Figure 2.7: Ackermann function

the next four deal with substitution. The final three rules are for duplication and erasing, where we use $\alpha$ to range over all other agents in the system.

There are three additional rules not in the figure, that we explain in more details, that implement iteration. When the iterator agent interacts with a closed abstraction we have the following rule:




This rule creates a new agent $It_c$ that will interact with numbers. The agent also holds on to the body and the variable edge of the abstraction. The two rules for the $It_c$ agent are as follows. The first rule is when we erase the function, and connect the result to the base value.




The final rule is when we unfold one level of iteration. Here the function is duplicated with $\delta$ agents, and one copy is applied to the base value as required. Because the function being duplicated is closed, the duplication process is easily proved to be correct.

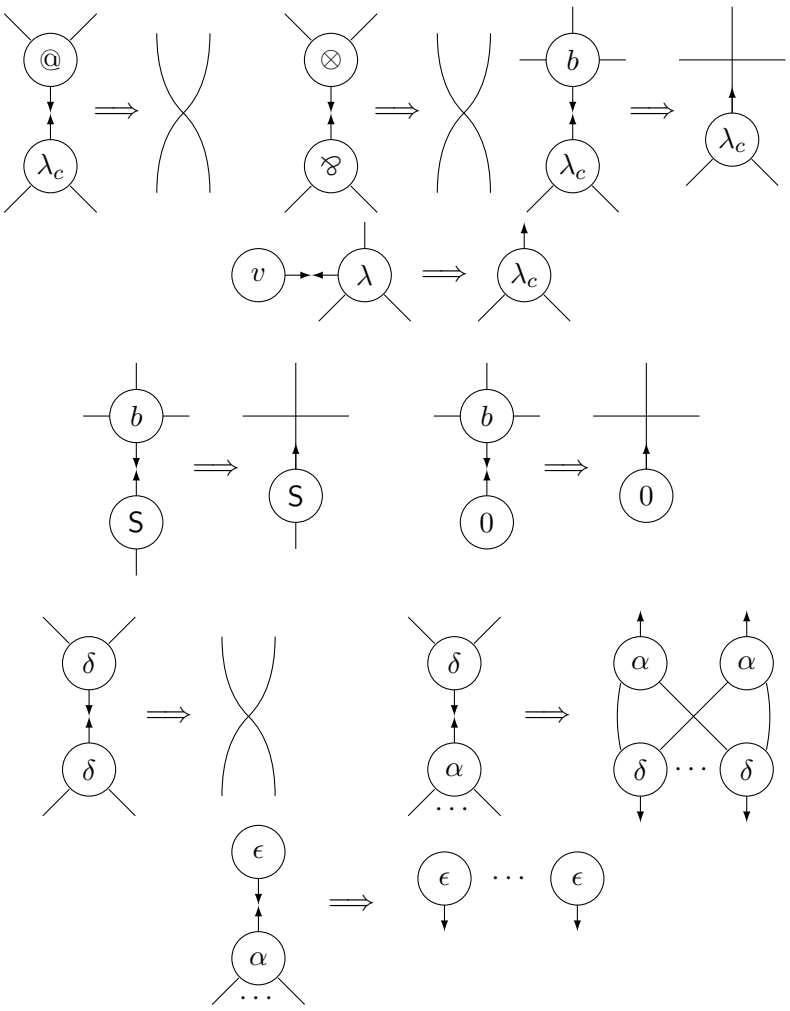
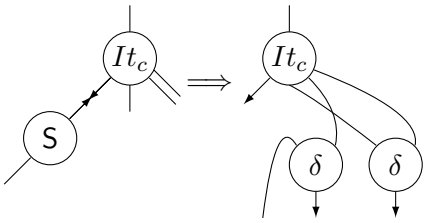



Figure 2.8: Interaction Rules




## 2.3 Summary

In this chapter, we reviewed two presentations of interaction nets: the original graphical calculus [36] and a textual calculus [17]. In the next chapter, we review interaction net evaluators based on these calculi.

We also presented several examples of interaction nets that give a flavour of how they are used. In particular, the examples of arithmetic operations will be used later as benchmark programs to compare evaluators and other major interpreters in Chapter 3, 4 and 7.

# Chapter 3

# Related works: evaluators towards efficient computation

In this chapter we review evaluators of interaction nets, focusing on efficiency. First, we review previous evaluators that focus on efficient computation. Next, explicating internal data structures in those evaluators, we introduce a standardised implementation model. Finally, we compare those methods in terms of efficient computation, especially execution time.

## 3.1 Overview

A number of evaluators for interaction nets have been developed, starting from Lafont [36]. A first attempt to build a programming language was Gay's interpreter [20] in 1991. This system provides an environment to develop interaction net programs with the typing system proposed by Lafont. In 2002, a graphical interpreter $\mathsf{in}^2$ was proposed by Lippi [38] and it showed an aspect of interaction nets as a visual programming tool, followed by INblobs [3].

Over the years, various interaction nets evaluators have been proposed – AMINE (and MPINE) [57] in 2000, PIN [28], INET [29], amineLight [30] and ingpu [34] in 2008, 2009, 2010, 2014 respectively. One goal of these evaluators was to produce efficient implementations of interaction nets. In the remainder of this section we review each of these evaluators in turn.

**AMINE** was proposed in 2000 [57] and it is the first evaluator based on the textual calculus of interaction nets [17]. In this evaluator, each variable in the calculus is repres-

ented as a node that has the same name as the variable, and the substitution operation for a variable requires a search to find the node which corresponds to the variables. To prevent searching those nodes deeply, *annotated terms*, that have name lists of terms as annotations, are introduced; however, processing these annotations consume a considerable amount of time during execution [30]. MPINE was also proposed as a concurrent computation version of AMINE.

**in$^2$** was proposed in 2002 [38] and it is based on Lafont's original graphical calculus [36]. It is written in the C language in order to perform more efficiently.

**PIN** was proposed in 2008 [28] and it is also based on the graphical calculus, so it is free from substitutions of variables. It works on a newly introduced abstract machine with its bytecodes, and the computation is done by executing sequences of the bytecodes. Thanks to the bytecode technology, it can perform three times faster than AMINE [28].

**INET** was proposed in 2009 [29] and it is considered as another version of PIN, therefore it is also based on the graphical calculus. Instead of using bytecodes, it translates the given nets into codes in the C language and compiles those. It can perform about six times faster than AMINE [29].

**amineLight** was proposed in 2010 [30] and it is based on a newly introduced textual calculus, called *lightweight calculus*, which is a refined version of the calculus [17]. This calculus shows all essential computation can be performed by using lightweight substitutions: $x = t$, $x = s \to t = s$ rather than using deep level substitutions: $x = t$, $u = s \to u[t/x] = s$. It can perform between 20 and 520 times faster than AMINE [30].

**ingpu** was proposed in 2014 [34] and it is also based on the lightweight calculus, which is a textual calculus. This evaluator was implemented by using a Graphical Processing Unit (GPU). One advantage of the textual calculi is locality in rewritings of interactions, and this evaluator performs a great deal of interactions in parallel. Re-wirings, however, are realised by substitutions that require synchronised rewritings, and this evaluator executes each substitution operation sequentially. Consequently, its performance is about the same as, or slightly lower than, amineLight [34].

All these evaluators are divided into two groups by the basis calculus: the graph rewriting system and textual calculi. The former group of evaluators are in$^2$, PIN and INET, and these evaluators have almost the same encoding methods of nets. The latter

group includes AMINE (MPINE), amineLight and ingpu. Our method proposed in this thesis belongs to this textual calculi group.

In this section, we review evaluators in the graph rewriting group and in the following section we review the textual calculi group. Finally, we compare the encoding methods in both groups in terms of efficient computation.

## 3.2 Evaluators based on the graph rewriting system

In this section we survey data-structures of nets in the evaluator $\mathsf{in}^2$ proposed by Lippi [38] which was introduced two years after AMINE was defined. We also survey the two evaluators PIN [28] and INET [29] which were proposed eight years after AMINE was defined.

With respect to the data-structure of nets, wires are represented as mutual connections of the appropriate ports. Rewritings of nets in those evaluators are based on the graph rewriting system, and they avoid having a complicated substitution problem that is caused in AMINE.

The main feature of PIN and INET is to give a compilation from nets into abstract machine codes and from nets into the C language respectively. Therefore, PIN and INET are compilers, whereas $\mathsf{in}^2$, AMINE and amineLight are interpreters which evaluate nets directly. INET is a successor version of PIN and experimental results [29] indicate that it can perform approximately between four and six times faster than AMINE, while PIN is about three times faster [28] than AMINE. The execution speed of $\mathsf{in}^2$ is not described in the paper [38].

In terms of the data-structures, PIN and INET use the same method to represent nets. The connection method of $\mathsf{in}^2$ is regarded as an alternative method to the one in PIN and INETS. First we review INET, and then we introduce $\mathsf{in}^2$ as an alternative net encoding method.

### 3.2.1 INET

In this section, we review the execution model of INET. Here we introduce a restricted version which deals with only the original graph rewriting system (without syntactic sugar).

A machine state in this model is defined by:

- a heap of agent nodes,
- a stack of active pairs,

- a rule table,
- a runtime environment.

**Heap of agent nodes** A heap is a memory model for agents, together with functions to manipulate the name table. Intuitively, an agent node `Agent`, the heap `Heap` and these runtime functions are written in the C language as:

```
/* for symbol unique numbers */
#define ID_NAME 0

/* agent nodes and the heap */
typedef struct Agent {
  int id;
  struct Port port[MAX_PORT];
} Agent;

typedef struct Port {
  int portNum;
  int agent;
} Port;

Agent Heap[];
```

where

- We assume that each symbol $\alpha_1, \ldots, \alpha_n$ for agents is allocated to a unique number $1, \ldots, n$, and these are stored with their arities $ar(\alpha_1), \ldots, ar(\alpha_n)$ into the array `Symbols` and `Arities` as follows:

```
#define ID_NAME 0
#define ID_α1 1
 ⋮
#define ID_αn n
#define MAX_AGENTID n
char Symbols[MAX_AGENTID+1] = {"", "α1",...,"αn"};
int Arities[MAX_AGENTID+1] = {1, ar(α1),...,ar(α1)};
```

- In the `Agent` structure,
  - an agent is defined by assigning a unique number for the agent to the `id`,
  - a name in the interface, whose arity is 1, is defined by assigning `ID_NAME` to the `id`.
- We assume, for simplicity, the following pre-defined constant that can be known during the compilation: `MAX_PORT` which gives the size of `Heap` and the maximum number of ports.

To allocate agent nodes, we define the following functions:

```
/* to allocate and de-allocate agents and the interface */
int mkAgent(int id);
int mkInterface();
void freeAgent(int a);
```

where

- the function `mkAgent` allocates an agent node and assigns the argument `id` to the attribute `id`. It returns an index for the allocated node in the `Heap`.
- the function `mkInterface` allocates an agent node and assigns `ID_NAME` to the attribute `id`. It returns an index for the allocated node in the `Heap`.
- The function `freeAgent` deallocates an agent from the Heap.

**Stack of active pairs** Next, we define functions to manage the stack of active pairs `ActivePairs` as:

```
typedef struct Active {
  int a1;
  int a2;
} Active;

Active ActivePairs[MAX_ACTIVE];

/* to manipulate ActivePairs */
void pushActive(int a1, int a2);
int popActive(int *a1, int *a2);
```

where

- we assume, for simplicity, a pre-defined constant: `MAX_ACTIVE`, which is the maximum size of `ActivePairs`,
- the function `pushActive` pushes the given arguments onto the stack `ActivePairs`,
- the function `popActive` pops values from `ActivePairs`. If it succeeds, then it substitutes the removed items for its arguments `a1` and `a2`, and returns `1`. If there is no entry, it just returns `0`.

To manipulate agent nodes, we define the following macro:

```
#define connect(a1, p1, a2, p2)    \
  Heap[a1].port[p1].agent   = a2;  \
  Heap[a1].port[p1].portNum = p2;  \
  Heap[a2].port[p2].agent   = a1;  \
  Heap[a2].port[p2].portNum = p1;  \
  if (p1==0 && p2==0) pushActive(a1, a2)

#define getPort(a, p) (Heap[a].port[p])
```

We note that every wire between agents is represented by mutual connections.

**Example 3.2.1**

*We can represent the net on the most left-hand side in Figure 2.1, which is presented as* $\mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(\mathtt{Z})$*, using the following set of codes:*

```
/* symbols */
#define ID_Add 1
#define ID_Z 2
#define ID_S 3
#define MAX_AGENTID 3
char Symbols[MAX_AGENTID+1] = {"", "Add", "Z", "S"};
int Arities[MAX_AGENTID+1] = {1, 2, 0, 1};

/* interface */
int r;

void mkNet() {
```

```
  r = mkInterface();

  /* Add(Z,r) */
  int aAdd = mkAgent(ID_Add);
  int aZ = mkAgent(ID_Z);
  connect(aAdd, 1, aZ, 0);
  connect(aAdd, 2, r, 1);

  /* S(Z) */
  int bS = mkAgent(ID_S);
  int bZ = mkAgent(ID_Z);
  connect(bS, 1, bZ, 0);

  /* Add(Z,r)=S(Z) */
  connect(aAdd, 0, bS, 0);
}
```

**Rule table** Next, we define the rule table `R` which stores function pointers according to the `id` attributes of agent nodes of active pairs. In this system, each interaction rule is performed by applying a function that is provided for each interaction rule. Here, for simplicity, we use the following symbol matrix while the original one is a hash table:

```
typedef void (*RuleFun)(int a1, int a2);
RuleFun R[MAX_AGENTID][MAX_AGENTID];
void initRuleTable();
```

These functions operate re-wiring of their arguments according to interaction rules. For example a function for the interaction rule $\texttt{Add}(x, x) \bowtie \texttt{Z}$ connects each agent that is connected to the ports of agent `Add` and makes entries for `Add` and `Z` free. This can be defined as follows:

```
void Add_Z(int a1, int a2) {
  connect(getPort(a1,1).agent, getPort(a1,1).portNum,
          getPort(a1,2).agent, getPort(a1,2).portNum);
  freeAgent(a1);
  freeAgent(a2);
}
```

A function for $\texttt{Add}(y, \texttt{S}(w)) \bowtie \texttt{S}(\texttt{Add}(y, w)))$ is defined as:

```
void Add_S(int a1, int a2) {
  int newS = mkAgent(ID_S);
  int newAdd = mkAgent(ID_Add);
  connect(newS, 1, newAdd, 2);
  connect(getPort(a1,1).agent, getPort(a1,1).portNum,
          newAdd, 1);
  connect(getPort(a1,2).agent, getPort(a1,2).portNum,
          newS, 0);
  connect(getPort(a2,1).agent, getPort(a2,1).portNum,
          newAdd, 0);
  freeAgent(a1);
  freeAgent(a2);
}
```

The function `initRuleTable` initialises the rule table `R`:

```
void errorPair(int a1, int a2) {
  printf("There is no rule for the active pair (%d,%d).\n", a1, a2);
  exit(-1);
}
void initRuleTable() {
  int i,j;
  for (i=0; i<= MAX_AGENTID; i++)
    for (j=0; j<= MAX_AGENTID; j++)
      R[i][j] = errorPair;
  /* interaction rules */
  R[ID_a][ID_b]=&a_b;
    ⋮
}
```

Assignments of functions for interaction rules to the set `R` are declared as exemplified below:

```
  /* interaction rules */
  R[ID_Add][ID_Z] = &Add_Z;
  R[ID_Add][ID_S] = &Add_S;
```

**Runtime environment** Next, we define runtime functions:

```
void eval();
void putsAgent(int a);
```

where

- the function `eval` runs through the active pair stack, pops an active pair and calls the appropriate rule function which rewrites the active pair. The `eval` function is defined as follows:

```
void eval() {
  int a1, a2;
  while (popActive(&a1, &a2)) {
    R[Heap[a1].id][Heap[a2].id](a1, a2);
  }
}
```

- the function `putsAgent` simply prints a net to the screen. It takes as an argument a location of an agent in the heap and pretty prints the net connected to the agent.

```
void putsAgent(int a) {
  int arity = Arities[Heap[a].id];
  printf("%s", Symbols[Heap[a].id]);
  if (arity != 0) {
    printf("(");
    int i;
    for (i=1; i<=arity; i++) {
      if (Heap[a].port[i].portNum == 0) {
        putsAgent(Heap[a].port[i].agent);
      } else {
        printf("AUX");
      }
      if (i<arity) printf(",");
    }
    printf(")");
  }
}
```

**Main function** The `main` function simply calls a set of pre-generated functions which build a net and evaluates it to normal form. First, the environment (the heap, the active pair stack and the rule table) is initialised by a call to a function `init`. Once the environment is initialised, the initial net is built using a function `mkNet` followed by the `eval` function which evaluates the net. Finally, the `putsAgent` function is called with a reference to the interface of the net (which is preserved throughout execution). The following is the main function of the net in Example 3.2.1:

```
int main() {
  init();
  mkNet();
  eval();
  putsAgent(Heap[r].port[1].agent);
  return 0;
}
```

### 3.2.2 in$^2$

in$^2$ uses the same encoding method for wires between auxiliary ports, and thus those are mutually connected. In contrast to PIN and INET, other sorts of connection are represented as single links, and thus a connection between a principal port and an auxiliary port is represented as a link from the auxiliary port to the principal port. Moreover, there is no connection information between principal ports on these agents nodes because this connection is managed by the active pair stack. Thus, the connection is defined as follows:

```
#define connect(a1, p1, a2, p2)       \
  if (p1 == 0 && p2 == 0) {           \
    pushActive(a1, a2);               \
  } else if (p1 == 0 && p2 != 0) {    \
    Heap[a2].port[p2].agent   = a1;   \
    Heap[a2].port[p2].portNum = p1;   \
  } else if (p1 != 0 && p2 == 0) {    \
    Heap[a1].port[p1].agent   = a2;   \
    Heap[a1].port[p1].portNum = p2;   \
  } else {                            \
    Heap[a1].port[p1].agent   = a2;   \
    Heap[a1].port[p1].portNum = p2;   \
```

```
  Heap[a2].port[p2].agent   = a1;   \
  Heap[a2].port[p2].portNum = p1;   \
}
```

## 3.3 Evaluators based on the textual calculi

In this section we review evaluators based on textual calculi, AMINE (MPINE) [57], amineLight [30] and ingpu [34] in 2000, 2010 and 2014 respectively.

In the textual calculi, rewritings in interaction nets are divided into two groups: interaction and re-wiring. The rewritings for the interaction can be performed locally thanks to names introduced for connections between auxiliary ports. The ingpu evaluator is implemented in parallel taking advantage of this property. The re-wiring is realised by substitution of names. To keep consistent relationship between names, some extra rules are required which potentially cause some overhead in the computation. Consequently, the performance of ingpu, although it can be performed in parallel, does not exceed the performance of amineLight.

In this section we survey the management methods for the names that have been introduced in these textual calculi. First, we review AMINE (MPINE) that is based on the textual calculus [17]. Next, we review a lightweight abstract machine of amineLight that is based on another textual calculus, called lightweight calculus [30] that is also the base calculus of ingpu.

### 3.3.1 AMINE (MPINE)

AMINE and MPINE were proposed by Pinto [57] in 2000. Those are based on the textual calculus [17]. MPINE is a concurrent version of AMINE.

First, we review the idea introduced in AMINE that avoids deep level searching for substitutions. In the term calculus active pairs as well as connection information are represented as equations. In order to bind fragments of connections such as $x = t$, we search for the other occurrence of $x$ (in the set of equations) and perform the substitution operation on the $x$. In AMINE, to perform this searching efficiently, the LHS and the RHS terms in equations are attached a sequence of names that occur in the term. This is written as ${}_{(\vec{x})}.t$ for the $t$ and the name sequence $\vec{x}$, and it is called an *annotated term*[1]. The name sequences are maintained correctly during rewritings. Thus, instead of searching all terms deeply, the name $x$ can be found at the top level.

[1] Originally, it is denoted as $\{\vec{x}\}.t$, but to avoid confusion of notation for sets, we write it as ${}_{(\vec{x})}.t$

The annotated term is defined as follows:

**Definition 3.3.1 (Annotated Terms)**

- Annotated terms $t_a$ *are built from* $\Sigma$ *and* $\mathcal{N}$ *using the following grammar:* $t_a ::= x \mid {}_{(\vec{x_a})}.t$, *where* $x$ *is a name,* $t$ *is a term built from* $\Sigma$ *and* $\mathcal{N}$, *and* $\vec{x_a}$ *is a sequence of names possibly containing the terminator symbol* $\boxtimes$. *The name sequence* $\vec{x_a}$ *is called an* annotation.

- *We use* $\mathcal{T}_a$ *as the set of annotated terms,* $t_a, s_a, u_a, \ldots$ *to range over annotated terms,* $\vec{t_a}, \vec{s_a}, \vec{u_a}, \ldots$ *to range over sequences of annotated terms, and "–" as the empty sequence.*

- *Given* $\vec{t_a} = t_{a_1}, \ldots, t_{a_k}$ *and* $\vec{s_a} = s_{a_1}, \ldots, s_{a_k}$, *we write* $(\vec{t_a}, \vec{s_a})$ *to denote the list* $(t_{a_1}, s_{a_1}), \ldots, (t_{a_k}, s_{a_k})$. *We use* $p_a, q_a, \ldots$ *to range over pairs of annotated terms,* $\vec{p_a}, \vec{q_a}, \ldots$ *to range over sequences of annotated terms pairs.*

- *We define function a function* Name *from a term into a sequence of names, and a function* $\mathsf{Ann}_t$ *from a term into an annotated term as follows:*

$$\begin{aligned} \mathsf{Name}(x) &\stackrel{\text{def}}{=} x, \\ \mathsf{Name}(\alpha(t_1, \ldots, t_n)) &\stackrel{\text{def}}{=} \mathsf{Name}(t_1), \ldots, \mathsf{Name}(t_n). \end{aligned}$$

$$\begin{aligned} \mathsf{Ann}_t(x) &\stackrel{\text{def}}{=} x, \\ \mathsf{Ann}_t(\alpha(t_1, \ldots, t_n)) &\stackrel{\text{def}}{=} {}_{(\mathsf{Name}(\alpha(t_1, \ldots, t_n)), \boxtimes)}.\alpha(t_1, \ldots, t_n). \end{aligned}$$

- *We extend the translation* $\mathsf{Ann}_t$ *into sequences of terms, term pairs, sequences of term pairs, equations and multisets of equations as follows:*

$$\begin{aligned} \mathsf{Ann}_{ts}(t_1, \ldots, t_n) &\stackrel{\text{def}}{=} \mathsf{Ann}_t(t_1), \ldots, \mathsf{Ann}_t(t_n). \\ \mathsf{Ann}_p(t_1, t_2) &\stackrel{\text{def}}{=} (\mathsf{Ann}_t(t_1), \mathsf{Ann}_t(t_2)). \\ \mathsf{Ann}_{ps}((s_1, u_1), \ldots, (s_n, u_n)) &\stackrel{\text{def}}{=} \mathsf{Ann}_p(s_1, u_1), \ldots, \mathsf{Ann}_p(s_n, u_n). \\ \mathsf{Ann}_e(t_1 = t_2) &\stackrel{\text{def}}{=} (\mathsf{Ann}_t(t_1), \mathsf{Ann}_t(t_2)). \\ \mathsf{Ann}_{es}(s_1 = u_1, \ldots, s_n = u_n) &\stackrel{\text{def}}{=} \mathsf{Ann}_e(s_1 = u_1), \ldots, \mathsf{Ann}_e(s_n = u_n). \end{aligned}$$

- *We write just* Ann *instead of* $\mathsf{Ann}_t$, $\mathsf{Ann}_{ts}$, $\mathsf{Ann}_p$, $\mathsf{Ann}_{ps}$, $\mathsf{Ann}_e$, $\mathsf{Ann}_{es}$ *when there is no ambiguity.*

**Abstract machine of AMINE** Here, we review the definition of the abstract machine formally:

**Definition 3.3.2 (Notations for maps)**

- *We use* [ ] *as an empty map.*

- *Let* $\psi$ *be a map. We use the following notation:*

$$\psi[x \mapsto a](z) \stackrel{\text{def}}{=} \begin{cases} a & (z \text{ is } x) \\ \psi(z) & (\text{otherwise}). \end{cases}$$

  *When* $\psi(x)$ *is undefined, we use the following notation:*

$$\psi[x \mapsto \bot](z) \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & (z = x) \\ \psi(z) & (\text{otherwise}). \end{cases}$$

  *We may also write* $\psi(x) = \bot$ *when* $\psi(x)$ *is undefined.*

**Definition 3.3.3 (Connection maps)**

- *We define a* heap E *as a map from a name into an annotated term.*

- *We define a* connection map P *as a map between names having the following property: if* $\mathrm{P}(x) = y$ *then* $\mathrm{P}(y) = x$*. We use the following notation:*

$$\mathrm{P}[x \leftrightarrow y](z) \stackrel{\text{def}}{=} \begin{cases} y & (z \text{ is } x) \\ x & (z \text{ is } y) \\ \mathrm{P}(z) & (\text{otherwise}). \end{cases}$$

  *When* $\mathrm{P}(x)$ *is undefined, we use the following notation:*

$$\mathrm{P}[x \leftrightarrow \bot](z) \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & (z = x) \\ \mathrm{P}(z) & (\text{otherwise}). \end{cases}$$

  - *We write* $\mathrm{P}_1\mathrm{P}_2$ *as the union map of* $\mathrm{P}_1$ *and* $\mathrm{P}_2$ *where a given argument is applied to* $\mathrm{P}_2$ *before* $\mathrm{P}_1$*.*
  - *We write* $\mathrm{P}^r$ *as the result of replacing each occurrence of a name* $x$ *for the domain and range of* P *with conserving their relationship by a fresh name* $x^r$ *respectively.*

**Definition 3.3.4 (Interaction operation)**

- Interaction rules in the abstract machine *have the form:* $(\alpha(\vec{s}), \beta(\vec{u}), \mathrm{P})$*. We require that there exists at most one rule for the same active pair, and each rule is closed under symmetry, thus if* $(t, s, \mathrm{P})$ *exists then* $(s, t, \mathrm{P})$ *also exists. In addition, we also require that there is no name which occurs in both* $\vec{s}$ *and* $\vec{u}$*.*

- *We use $\mathcal{R}_m$ as a set of interaction rules in the abstract machine. To write a rule corresponding to $r \in \mathcal{R}$, we need to split each linear name $x$ in $r$ into fresh names $x_1$ and $x_2$ respectively and store their linking information as $\mathrm{P}[x_1 \leftrightarrow x_2]$. We denote a rule in the abstract machine for $r \in \mathcal{R}$ as $\mathsf{Compile}(r)$, and a rule set for $\mathcal{R}$ as $\mathsf{Compile}(\mathcal{R})$.*

- *We define the interaction operations in the abstract machine as follows:*

$$\mathsf{Interaction}(\alpha(\vec{s_1}),\, \beta(\vec{u_1})) \stackrel{\text{def}}{=} \begin{cases} (\mathsf{Ann}((\vec{s_1}, \vec{s^r}), (\vec{u_1}, \vec{u^r})),\, \mathrm{P}^r) & \\ \qquad\qquad (\text{when } (\alpha(\vec{s}),\, \beta(\vec{u}),\, \mathrm{P}) \in \mathcal{R}_m) & \\ (-, [\,]) \qquad\qquad (\text{otherwise}) & \end{cases}$$

*where $\vec{s^r}$, $\vec{u^r}$, $\mathrm{P}^r$ are the result of replacing each occurrence of a name $x$ for $\vec{s}$, $\vec{u}$ and* P, *conserving their original relationship respectively.*

$$\mathsf{Error}((_{(\vec{x_a})}.\alpha(\vec{s}),\, _{(\vec{y_a})}.\beta(\vec{u}))) \stackrel{\text{def}}{=} \begin{cases} - \qquad (\text{when } (\alpha(\vec{s}),\, \beta(\vec{u}),\, \mathrm{P}) \in \mathcal{R}_m) & \\ (_{(\vec{x_a})}.\alpha(\vec{s}),\, _{(\vec{y_a})}.\beta(\vec{u})) \qquad (\text{otherwise}) & \end{cases}$$

**Definition 3.3.5 (Machine configuration)**

*We define a configuration of the abstract machine state as a tuple having the following form:*

$$(\,\mathrm{E} \mid \mathrm{P} \mid S \mid V \mid C_y \mid op\,),$$

*where*

- E *is a heap,*
- P *is a connection map,*
- *$S$ is a sequence of annotated terms pairs representing equations,*
- *$V$ is a sequence of annotated terms representing an interface,*
- *$C_y$ is a sequence of annotated terms pairs representing a sequence of cycles, and pairs for which there is no interaction rule, and*
- *op is an instruction of this machine defined as follows:*

| *Instruction* | *Description* |
|---|---|
| `process`$(t_a, u_a)$ | *process the pair $(t_a, u_a)$,* |
| `delist` | *pop a pair from $S$ to be processed,* |
| `enlist`$((\vec{t_a}, \vec{u_a}))$ | *push pairs $(\vec{t_a}, \vec{u_a})$ onto $S$,* |
| `cycle`$(x, t_a)$ | *push a cycle pair $(x, t_a)$ to $C_y$.* |

*The abstract machine is loaded with* $op = \texttt{delist}$*, and stops when* $S$ *is the empty sequence and* $op$ *is* `delist`*. In Figure 3.1 and 3.2 we give the semantics of the machine as a set of transition rules of the form:*

$$(\,\mathrm{E} \mid \mathrm{P} \mid S \mid V \mid C_y \mid op\,) \Longrightarrow (\,\mathrm{E}' \mid \mathrm{P}' \mid S' \mid V' \mid C'_y \mid op'\,).$$

*For readability purposes, we present the transitions in a table format. For example, the entry:*

| | | Before | After |
|---|---|---|---|
| II.2 | Heap | $\mathrm{E}\ [z \mapsto t_a]$ | E |
| | Connection | $\mathrm{P}\ [x \leftrightarrow z]$ | P |
| | op | $\texttt{process}(x, y)$ | $\texttt{process}(t_a, y)$ |

*corresponds to:*

$$(\,\mathrm{E}[z \mapsto t_a] \mid \mathrm{P}[x \leftrightarrow z] \mid S \mid V \mid C_y \mid \texttt{process}(x, y)\,) \Longrightarrow (\,\mathrm{E} \mid \mathrm{P} \mid S \mid V \mid C_y \mid \texttt{process}(t_a, y)\,).$$

| | | Before | After |
|---|---|---|---|
| I | Connection | P | PP′ |
| | Cycles | $C_y$ | $\vec{p_a}$, $C_y$ |
| | op | $\texttt{process}(_{(\vec{x_a})}.\alpha(\vec{s}),\ _{(\vec{y_a})}.\beta(\vec{u}))$ | $\texttt{enlist}(\vec{q_a})$ |

where $\mathsf{Error}(_{(\vec{x_a})}.\alpha(\vec{s}),\ _{(\vec{y_a})}.\beta(\vec{u})) = \vec{p_a}$,

$\mathsf{Interaction}(\alpha(\vec{s}),\ \beta(\vec{u})) = (\vec{q_a},\ \mathrm{P}')$.

| | | | |
|---|---|---|---|
| II.1 | Connection | $\mathrm{P}\ [x \leftrightarrow y]$ | $\mathrm{P}\ [x \leftrightarrow y]$ |
| | op | $\texttt{process}(x, y)$ | $\texttt{cycle}(x, y)$ |
| II.2 | Heap | $\mathrm{E}\ [z \mapsto {}_{(\vec{x_a})}.\alpha(\vec{t})]$ | E |
| | Connection | $\mathrm{P}\ [x \leftrightarrow z]$ | P |
| | op | $\texttt{process}(x, y)$ | $\texttt{process}(_{(\vec{x_a})}.\alpha(\vec{t}),\ y)$ |
| II.3 | Heap | $\mathrm{E}\ [z \mapsto \bot]\ [w \mapsto {}_{(\vec{x_a})}.t]$ | E |
| | Connection | $\mathrm{P}\ [x \leftrightarrow z]\ [y \leftrightarrow w]$ | $\mathrm{P}\ [x \leftrightarrow z]$ |
| | op | $\texttt{process}(x, y)$ | $\texttt{process}(x,\ _{(\vec{x_a})}.t)$ |
| II.4 | Heap | $\mathrm{E}\ [z \mapsto \bot]\ [w \mapsto \bot]$ | E |
| | Connection | $\mathrm{P}\ [x \leftrightarrow z]\ [y \leftrightarrow w]$ | $\mathrm{P}\ [z \leftrightarrow w]$ |
| | op | $\texttt{process}(x, y)$ | `delist` |

Figure 3.1: Transitions for codes $\texttt{process}(_{(\vec{x_a})}.\alpha(\vec{s}),\ _{(\vec{y_a})}.\beta(\vec{u}))$ and $\texttt{process}(x, y)$

| | | Before | After |
|---|---|---|---|
| III.0 | op | $\texttt{process}(_{(\vec{x_a})}.t,\ y)$ | $\texttt{process}(y,\ _{(\vec{x_a})}.t)$ |
| III.1 | Heap | E $[x \mapsto {}_{(\vec{x},\boxtimes)}.t]$ | E |
| | Connection | P $[x \leftrightarrow y]$ | P |
| | op | $\texttt{process}(z,\ _{(y,\vec{y_a})}.s)$ | $\texttt{process}(z,\ _{(\vec{x},\vec{y_a})}.s[t/y])$ |
| III.2 | Heap | E $[x \mapsto \bot]$ | E |
| | Connection | P $[x \leftrightarrow y]$ | P $[x \leftrightarrow y]$ |
| | op | $\texttt{process}(z,\ _{(y,\vec{y_a})}.s)$ | $\texttt{process}(z,\ _{(\vec{y_a},y)}.s)$ |
| III.3 | Connection | P $[x \leftrightarrow y]$ | P $[x \leftrightarrow y]$ |
| | op | $\texttt{process}(x,\ _{(y,\vec{y_a})}.s)$ | $\texttt{cycle}(x,\ _{(y,\vec{y_a})}.s)$ |
| III.4 | Heap | E $[x \mapsto {}_{(\vec{x_a})}.t]$ | E |
| | Connection | P $[x \leftrightarrow z]$ | P |
| | op | $\texttt{process}(z,\ _{(\boxtimes,\vec{y_a})}.s)$ | $\texttt{process}(_{(\vec{x_a})}.t,\ _{(\boxtimes,\vec{y_a})}.s)$ |
| III.5 | Heap | E $[x \mapsto \bot]$ | E $[z \mapsto {}_{(\vec{y_a},\boxtimes)}.s]$ |
| | Connection | P $[x \leftrightarrow z]$ | P $[x \leftrightarrow z]$ |
| | op | $\texttt{process}(z,\ _{(\boxtimes,\vec{y_a})}.s)$ | $\texttt{delist}$ |
| T.1 | Pairs | $(t_a, u_a),\ S$ | $S$ |
| | op | $\texttt{delist}$ | $\texttt{process}(t_a, u_a)$ |
| T.2 | Pairs | $S$ | $(t_a, u_a),\ S$ |
| | op | $\texttt{enlist}((t_a, u_a),\ \vec{p})$ | $\texttt{enlist}(\vec{p})$ |
| T.3 | op | $\texttt{enlist}(-)$ | $\texttt{delist}$ |
| T.4 | Cycles | $C_y$ | $(t_a, u_a),\ C_y$ |
| | op | $\texttt{cycle}(t_a, u_a)$ | $\texttt{delist}$ |

Figure 3.2: Transitions for other codes

**Definition 3.3.6 (Initial loading)**

- *We define a translation* Compile() *from a configuration of interaction nets into a configuration of the abstract machine state as follows:*

$$\mathsf{Compile}(\langle\ \vec{t}\ \mid\ \Delta\ \rangle) \stackrel{\text{def}}{=} (\ [\,] \mid \mathrm{P}' \mid \mathsf{Ann}(\Delta') \mid \mathsf{Ann}(\vec{t'}) \mid - \mid \texttt{delist}\ )$$

*where $\vec{t'}$ and $\Delta'$ are the result of splitting each occurrence of a linear name $x$ in $\vec{t}$ and $\Delta$ into fresh names $x_1$ and $x_2$ respectively, and $\mathrm{P}'$ is the result of storing their linking information $[x_1 \leftrightarrow x_2]$.*

We can obtain the computation result by using the following operation:

**Definition 3.3.7 (Updating operation)**

- *We define a function* $\mathsf{remAnn} : \mathcal{T} \cup \mathcal{T}_a \to \mathcal{T}$ *to obtain a term without annotation as follows:*

$$\begin{cases} \mathsf{remAnn}(x) & \stackrel{\text{def}}{=} x, \\ \mathsf{remAnn}(\alpha(t_1, \ldots, t_n)) & \stackrel{\text{def}}{=} \alpha(\mathsf{remAnn}(t_1), \ldots, \mathsf{remAnn}(t_n)), \\ \mathsf{remAnn}(_{(\vec{x_a})}.t) & \stackrel{\text{def}}{=} \mathsf{remAnn}(t). \end{cases}$$

*We extend this function into sequences as follows:*

$$\mathsf{remAnn}(t_1, \ldots, t_n) \stackrel{\text{def}}{=} \mathsf{remAnn}(t_1), \ldots, \mathsf{remAnn}(t_n).$$

- *We define the updating operation for distributed information* Update *as follows:*

$$\begin{array}{ll} \mathsf{Update}(\,\mathrm{E} \mid \mathrm{P} \mid S \mid V \mid C_y \mid op\,) & \stackrel{\text{def}}{=} \mathsf{Collect}(\mathrm{E} \mid \mathrm{P} \mid V), \\ \mathsf{Collect}(\mathrm{E} \mid \mathrm{P}[x \leftrightarrow y] \mid V) & \stackrel{\text{def}}{=} \mathsf{Collect}(\mathrm{E}[y/x] \mid \mathrm{P} \mid V[y/x]), \\ \mathsf{Collect}(\mathrm{E}[x \mapsto {}_{(\vec{x_a})}.t] \mid [\,] \mid V) & \stackrel{\text{def}}{=} \mathsf{Collect}(\mathrm{E} \mid [\,] \mid V[t/x]), \\ \mathsf{Collect}([\,] \mid [\,] \mid V) & \stackrel{\text{def}}{=} \mathsf{remAnn}(V). \end{array}$$

**Example 3.3.8**

*Let us consider the case for Example 2.1.9. First, we obtain rules in the machine as follows:*

$$\begin{cases} (\mathtt{Add}(y_1, \mathtt{S}(w_1)), \mathtt{S}(\mathtt{Add}(y_2, w_2)),\ [y_1 \leftrightarrow y_2][w_1 \leftrightarrow w_2]) \\ (\mathtt{Add}(y_1, y_2), \mathtt{Z},\ [y_1 \leftrightarrow y_2]). \end{cases}$$

*For the configuration* $\langle\ r \mid \mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(\mathtt{Z})\ \rangle$*, we obtain the following initial machine state:*

$$(\,[\,] \mid [r_1 \leftrightarrow r_2] \mid (\mathtt{Add}(\mathtt{Z}, r_2),\ \mathtt{S}(\mathtt{Z})) \mid r_1 \mid - \mid \mathtt{delist}\,).$$

*The execution result is given below:*

$$\left(\ [\,] \,\middle|\, [r_1 \leftrightarrow r_2] \,\middle|\, (\mathtt{Add}(\mathtt{Z}, r_2),\ \mathtt{S}(\mathtt{Z})) \,\middle|\, r_1 \,\middle|\, - \,\middle|\, \mathtt{delist}\ \right)$$

$\Longrightarrow_{T.3}$

$$\left(\ [\,] \,\middle|\, [r_1 \leftrightarrow r_2] \,\middle|\, - \,\middle|\, r_1 \,\middle|\, - \,\middle|\, \mathtt{process}(\mathtt{Add}(\mathtt{Z}, r_2), \mathtt{S}(\mathtt{Z}))\ \right)$$

$\Longrightarrow_{I}$

$$\left(\ [\,] \,\middle|\, \begin{array}{l} [y_1' \leftrightarrow y_2'] \\ [w_1' \leftrightarrow w_2'] \\ [r_1 \leftrightarrow r_2] \end{array} \,\middle|\, - \,\middle|\, r_1 \,\middle|\, - \,\middle|\, \mathtt{enlist} \left( \begin{array}{l} (_{(\boxtimes)}.\mathtt{Z},\ y_1'), \\ (r_2,\ _{(w_1', \boxtimes)}.\mathtt{S}(w_1')), \\ (_{(\boxtimes)}.\mathtt{Z},\ _{(y_2', w_2', \boxtimes)}.\mathtt{Add}(y_2', w_2')) \end{array} \right)\ \right)$$

$\Longrightarrow^{*}_{T.2}\Longrightarrow_{T.3}$

$$\left(\ [\,] \,\middle|\, \begin{array}{l} [y_1' \leftrightarrow y_2'] \\ [w_1' \leftrightarrow w_2'] \\ [r_1 \leftrightarrow r_2] \end{array} \,\middle|\, \begin{array}{l} (_{(\boxtimes)}.\mathtt{Z},\ y_1'), \\ (r_2,\ _{(w_1', \boxtimes)}.\mathtt{S}(w_1')), \\ (_{(\boxtimes)}.\mathtt{Z},\ _{(y_2', w_2', \boxtimes)}.\mathtt{Add}(y_2', w_2')) \end{array} \,\middle|\, r_1 \,\middle|\, - \,\middle|\, \mathtt{delist}\ \right)$$

$\Longrightarrow_{T.1}$

$$\left(\begin{array}{c|c|c|c|c} [\,] & \begin{array}{l}[y_1' \leftrightarrow y_2'] \\ [w_1' \leftrightarrow w_2'] \\ [r_1 \leftrightarrow r_2]\end{array} & \begin{array}{l}(r_2,\ {}_{(w_1',\boxtimes)}.\mathtt{S}(w_1')), \\ ({}_{(\boxtimes)}.\mathtt{Z},\ {}_{(y_2',w_2',\boxtimes)}.\mathtt{Add}(y_2', w_2'))\end{array} & r_1 & - \ \mathtt{process}({}_{(\boxtimes)}.\mathtt{Z},\ y_1') \end{array}\right)$$

$\Longrightarrow_{III.0}$

$$\left(\begin{array}{c|c|c|c|c} [\,] & \begin{array}{l}[y_1' \leftrightarrow y_2'] \\ [w_1' \leftrightarrow w_2'] \\ [r_1 \leftrightarrow r_2]\end{array} & \begin{array}{l}(r_2,\ {}_{(w_1',\boxtimes)}.\mathtt{S}(w_1')), \\ ({}_{(\boxtimes)}.\mathtt{Z},\ {}_{(y_2',w_2',\boxtimes)}.\mathtt{Add}(y_2', w_2'))\end{array} & r_1 & - \ \mathtt{process}(y_1',\ {}_{(\boxtimes)}.\mathtt{Z}) \end{array}\right)$$

$\Longrightarrow_{III.5}$

$$\left(\begin{array}{c|c|c|c|c} [y_1' \mapsto {}_{(\boxtimes)}.\mathtt{Z}] & \begin{array}{l}[y_1' \leftrightarrow y_2'] \\ [w_1' \leftrightarrow w_2'] \\ [r_1 \leftrightarrow r_2]\end{array} & \begin{array}{l}(r_2,\ {}_{(w_1',\boxtimes)}.\mathtt{S}(w_1')), \\ ({}_{(\boxtimes)}.\mathtt{Z},\ {}_{(y_2',w_2',\boxtimes)}.\mathtt{Add}(y_2', w_2'))\end{array} & r_1 & - \ \mathtt{delist} \end{array}\right)$$

$\Longrightarrow_{T.1}$

$$\left(\begin{array}{c|c|c|c|c} [y_1' \mapsto {}_{(\boxtimes)}.\mathtt{Z}] & \begin{array}{l}[y_1' \leftrightarrow y_2'] \\ [w_1' \leftrightarrow w_2'] \\ [r_1 \leftrightarrow r_2]\end{array} & ({}_{(\boxtimes)}.\mathtt{Z},\ {}_{(y_2',w_2',\boxtimes)}.\mathtt{Add}(y_2', w_2')) & r_1 & - \ \begin{array}{l}\mathtt{process}(r_2, \\ \quad {}_{(w_1',\boxtimes)}.\mathtt{S}(w_1'))\end{array} \end{array}\right)$$

$\Longrightarrow_{III.2}$

$$\left(\begin{array}{c|c|c|c|c} [y_1' \mapsto {}_{(\boxtimes)}.\mathtt{Z}] & \begin{array}{l}[y_1' \leftrightarrow y_2'] \\ [w_1' \leftrightarrow w_2'] \\ [r_1 \leftrightarrow r_2]\end{array} & ({}_{(\boxtimes)}.\mathtt{Z},\ {}_{(y_2',w_2',\boxtimes)}.\mathtt{Add}(y_2', w_2')) & r_1 & - \ \begin{array}{l}\mathtt{process}(r_2, \\ \quad {}_{(\boxtimes,w_1')}.\mathtt{S}(w_1'))\end{array} \end{array}\right)$$

$\Longrightarrow_{III.5}$

$$\left(\begin{array}{c|c|c|c|c} \begin{array}{l}[r_2 \mapsto {}_{(w_1',\boxtimes)}.\mathtt{S}(w_1')] \\ [y_1' \mapsto {}_{(\boxtimes)}.\mathtt{Z}]\end{array} & \begin{array}{l}[y_1' \leftrightarrow y_2'] \\ [w_1' \leftrightarrow w_2'] \\ [r_1 \leftrightarrow r_2]\end{array} & ({}_{(\boxtimes)}.\mathtt{Z},\ {}_{(y_2',w_2',\boxtimes)}.\mathtt{Add}(y_2', w_2')) & r_1 & - \ \mathtt{delist} \end{array}\right)$$

$\Longrightarrow_{T.1}$

$$\left(\begin{array}{c|c|c|c|c} \begin{array}{l}[r_2 \mapsto {}_{(w_1',\boxtimes)}.\mathtt{S}(w_1')] \\ [y_1' \mapsto {}_{(\boxtimes)}.\mathtt{Z}]\end{array} & \begin{array}{l}[y_1' \leftrightarrow y_2'] \\ [w_1' \leftrightarrow w_2'] \\ [r_1 \leftrightarrow r_2]\end{array} & - & r_1 & - \ \begin{array}{l}\mathtt{process}({}_{(\boxtimes)}.\mathtt{Z}, \\ \quad {}_{(y_2',w_2',\boxtimes)}.\mathtt{Add}(y_2', w_2'))\end{array} \end{array}\right)$$

$\Longrightarrow_{I}$

$$\left(\begin{array}{c|c|c|c|c} \begin{array}{l}[r_2 \mapsto {}_{(w_1',\boxtimes)}.\mathtt{S}(w_1')] \\ [y_1' \mapsto {}_{(\boxtimes)}.\mathtt{Z}]\end{array} & \begin{array}{l}[y_1'' \leftrightarrow y_2''], \\ [y_1' \leftrightarrow y_2'] \\ [w_1' \leftrightarrow w_2'] \\ [r_1 \leftrightarrow r_2]\end{array} & - & r_1 & - \ \mathtt{enlist}\left(\begin{array}{l}(y_2', y_1''), \\ (w_2', y_2'')\end{array}\right) \end{array}\right)$$

$\Longrightarrow^{*}_{T.2}\Longrightarrow_{T.3}$

$$\left(\begin{array}{c|c|c|c|c} \begin{array}{l}[r_2 \mapsto {}_{(w_1',\boxtimes)}.\mathtt{S}(w_1')] \\ [y_1' \mapsto {}_{(\boxtimes)}.\mathtt{Z}]\end{array} & \begin{array}{l}[y_1'' \leftrightarrow y_2''], \\ [y_1' \leftrightarrow y_2'] \\ [w_1' \leftrightarrow w_2'] \\ [r_1 \leftrightarrow r_2]\end{array} & \begin{array}{l}(y_2', y_1''), \\ (w_2', y_2'')\end{array} & r_1 & - \ \mathtt{delist} \end{array}\right)$$

$\Longrightarrow_{T.1}$

$$\left(\begin{array}{l|l|l|l|l|l}
[r_2 \mapsto_{(w_1',\boxtimes)} \mathsf{S}(w_1')] & [y_1'' \leftrightarrow y_2''], & (w_2', y_2'') & r_1 & - & \texttt{process}(y_2', y_1'') \\
[y_1' \mapsto_{(\boxtimes)} \mathsf{Z}] & [y_1' \leftrightarrow y_2'] & & & & \\
 & [w_1' \leftrightarrow w_2'] & & & & \\
 & [r_1 \leftrightarrow r_2] & & & &
\end{array}\right)$$

$\Longrightarrow_{II.4}$

$$\left(\begin{array}{l|l|l|l|l|l}
[r_2 \mapsto_{(w_1',\boxtimes)} \mathsf{S}(w_1')] & [y_1' \leftrightarrow y_2''] & (w_2', y_2'') & r_1 & - & \texttt{delist} \\
[y_1' \mapsto_{(\boxtimes)} \mathsf{Z}] & [w_1' \leftrightarrow w_2'] & & & & \\
 & [r_1 \leftrightarrow r_2] & & & &
\end{array}\right)$$

$\Longrightarrow_{T.1}$

$$\left(\begin{array}{l|l|l|l|l|l}
[r_2 \mapsto_{(w_1',\boxtimes)} \mathsf{S}(w_1')] & [y_1' \leftrightarrow y_2''] & - & r_1 & - & \texttt{process}(w_2', y_2'') \\
[y_1' \mapsto_{(\boxtimes)} \mathsf{Z}] & [w_1' \leftrightarrow w_2'] & & & & \\
 & [r_1 \leftrightarrow r_2] & & & &
\end{array}\right)$$

$\Longrightarrow_{II.4}$

$$\left(\begin{array}{l|l|l|l|l|l}
[r_2 \mapsto_{(w_1',\boxtimes)} \mathsf{S}(w_1')] & [w_1' \leftrightarrow y_1'] & - & r_1 & - & \texttt{delist} \\
[y_1' \mapsto_{(\boxtimes)} \mathsf{Z}] & [r_1 \leftrightarrow r_2] & & & &
\end{array}\right).$$

*Finally, by using the operation* Update*, we obtain the computation result as follows:*

$$\mathsf{Update}\left(\begin{array}{l|l|l|l|l|l}
[r_2 \mapsto_{(w_1',\boxtimes,)} \mathsf{S}(w_1')] & [w_1' \leftrightarrow y_1'] & - & r_1 & - & \texttt{delist} \\
[y_1' \mapsto_{(\boxtimes)} \mathsf{Z}] & [r_1 \leftrightarrow r_2] & & & &
\end{array}\right)$$

$$= \mathsf{Collect}\left(\begin{array}{l|l|l}
[r_2 \mapsto_{(y_1',\boxtimes)} \mathsf{S}(y_1')] & [r_1 \leftrightarrow r_2] & r_1 \\
[y_1' \mapsto_{(\boxtimes)} \mathsf{Z}] & &
\end{array}\right)$$

$$= \mathsf{Collect}\left(\begin{array}{l|l|l}
[r_2 \mapsto_{(y_1',\boxtimes)} \mathsf{S}(y_1')] & [\,] & r_2 \\
[y_1' \mapsto_{(\boxtimes)} \mathsf{Z}] & &
\end{array}\right)$$

$$= \mathsf{Collect}\left(\begin{array}{l|l|l} [y_1' \mapsto_{(\boxtimes)} \mathsf{Z}] & [\,] & \mathsf{S}(y_1') \end{array}\right)$$

$$= \mathsf{Collect}\left(\begin{array}{l|l|l} [\,] & [\,] & \mathsf{S}(\mathsf{Z}) \end{array}\right)$$

$$= \mathsf{S}(\mathsf{Z}).$$

**Abstract machine of MPINE** MPINE is a concurrent version of AMINE. The key feature is that the number of instructions can increase to more than one according to the number of threads.

**Definition 3.3.9 (Multi-thread machine configuration)**

*We define an n-threads configuration of the abstract machine state as a tuple having the following form:*

$$(\,\mathrm{E} \mid \mathrm{P} \mid S \mid V \mid C_y \mid op_1, \ldots, op_n\,)$$

*where*

- $op_1, \ldots, op_n$ *is a sequence of instructions,*
- *all other components are the same as before.*

**Definition 3.3.10 (Concurrent transition)**

*We define the transition* $\overset{ct}{\Longrightarrow}$ *of the n-threads configuration of the machine as follows:*

$$\frac{(\,\mathrm{E} \mid \mathrm{P} \mid S \mid V \mid C_y \mid op_i\,) \Longrightarrow (\,\mathrm{E}' \mid \mathrm{P}' \mid S' \mid V' \mid C'_y \mid op'_i\,)}{(\,\mathrm{E} \mid \mathrm{P} \mid S \mid V \mid C_y \mid op_1, ..., op_i, ..., op_n\,) \overset{ct}{\Longrightarrow} (\,\mathrm{E}' \mid \mathrm{P}' \mid S' \mid V' \mid C'_y \mid op_1, ..., op'_i, ..., op_n\,)}$$

*where the index* $i$ *is selected non-deterministically.*

Intuitively, this definition is intended to work by using a thread pool with shared data structures as follows:

- E, P, $S$, $C_y$ are implemented on a shared memory area,
- each thread in the thread pool performs each instruction $op_i$, according to the transition rules,
- by means of a synchronisation mechanism, it is ensured that there are no two threads which access to the shared memory for $S$, $C_y$ and both of E, P.

Next, we review an example such that the configuration may become inconsistent during the $\overset{ct}{\Longrightarrow}$-transition, even if some synchronisation mechanism avoids accessing shared memory at once. Here, we take an example [57] such that a consistent configuration may end in inconsistency. First, we look at the case where the configuration ends in consistency:

$$\begin{aligned}
&(\,[\,] \mid [x \leftrightarrow z][y \leftrightarrow w] \mid (x, {}_{(w,\boxtimes)}.t),\ (y, {}_{(z,\boxtimes)}.u),\ S \mid V \mid C_y \mid \texttt{delist}, \texttt{delist}\,)\\
&\overset{ct}{\Longrightarrow}{}^{*}_{T.1} \quad (\,[\,] \mid [x \leftrightarrow z][y \leftrightarrow w] \mid S \mid V \mid C_y \mid \texttt{process}(x, {}_{(w,\boxtimes)}.t), \texttt{process}(y, {}_{(z,\boxtimes)}.u)\,)\\
&\overset{ct}{\Longrightarrow}_{III.2} \quad (\,[\,] \mid [x \leftrightarrow z][y \leftrightarrow w] \mid S \mid V \mid C_y \mid \texttt{process}(x, {}_{(\boxtimes,w)}.t), \texttt{process}(y, {}_{(z,\boxtimes)}.u)\,)\\
&\overset{ct}{\Longrightarrow}_{III.5} \quad (\,[x \mapsto {}_{(\boxtimes,w)}.t] \mid [x \leftrightarrow z][y \leftrightarrow w] \mid S \mid V \mid C_y \mid \texttt{delist}, \texttt{process}(y, {}_{(z,\boxtimes)}.u)\,)\\
&\overset{ct}{\Longrightarrow}_{III.1} \quad (\,[\,] \mid [y \leftrightarrow w] \mid S \mid V \mid C_y \mid \texttt{delist}, \texttt{process}(y, {}_{(w,\boxtimes)}.u[t/x])\,)\\
&\overset{ct}{\Longrightarrow}_{III.2} \quad (\,[\,] \mid [y \leftrightarrow w] \mid S \mid V \mid C_y \mid \texttt{delist}, \texttt{process}(y, {}_{(\boxtimes,w)}.u[t/x])\,)\\
&\overset{ct}{\Longrightarrow}_{III.5} \quad (\,[y \mapsto {}_{(w,\boxtimes)}.u[t/x]] \mid [y \leftrightarrow w] \mid S \mid V \mid C_y \mid \texttt{delist}, \texttt{delist}\,).
\end{aligned}$$

Since the order of the application of the $\overset{ct}{\Longrightarrow}$-transition is non-deterministic, the configuration can end without the substitution for $x$ as follows:

$$
\begin{aligned}
&(\,[\,] \mid [x \leftrightarrow z][y \leftrightarrow w] \mid (x, {}_{(w,\boxtimes)}.t),\ (y, {}_{(z,\boxtimes)}.u),\ S \mid V \mid C_y \mid \texttt{delist}, \texttt{delist}\,) \\
&\overset{ct}{\Longrightarrow}{}^{*}_{T.1} \quad (\,[\,] \mid [x \leftrightarrow z][y \leftrightarrow w] \mid S \mid V \mid C_y \mid \texttt{process}(x, {}_{(w,\boxtimes)}.t), \texttt{process}(y, {}_{(z,\boxtimes)}.u)\,) \\
&\overset{ct}{\Longrightarrow}_{III.2} \quad (\,[\,] \mid [x \leftrightarrow z][y \leftrightarrow w] \mid S \mid V \mid C_y \mid \texttt{process}(x, {}_{(\boxtimes,w)}.t), \texttt{process}(y, {}_{(z,\boxtimes)}.u)\,) \\
&\overset{ct}{\Longrightarrow}_{III.2} \quad (\,[\,] \mid [x \leftrightarrow z][y \leftrightarrow w] \mid S \mid V \mid C_y \mid \texttt{process}(x, {}_{(\boxtimes,w)}.t), \texttt{process}(y, {}_{(\boxtimes,z)}.u)\,) \\
&\overset{ct}{\Longrightarrow}_{III.5} \quad (\,[x \mapsto {}_{(w,\boxtimes)}.t] \mid [x \leftrightarrow z][y \leftrightarrow w] \mid S \mid V \mid C_y \mid \texttt{delist}, \texttt{process}(y, {}_{(\boxtimes,z)}.u)\,) \\
&\overset{ct}{\Longrightarrow}_{III.5} \quad (\,[x \mapsto {}_{(w,\boxtimes)}.t][y \mapsto {}_{(z,\boxtimes)}.u] \mid [x \leftrightarrow z][y \leftrightarrow w] \mid S \mid V \mid C_y \mid \texttt{delist}, \texttt{delist}\,).
\end{aligned}
$$

This is because the traversing process for annotations, which is performed by applying rule III.2, supposes a condition such that both of the heap and connection map should be kept until the rule III.4 or III.5 is applied in order to finish the traversing. This condition may not hold in the $\overset{ct}{\Longrightarrow}$-transition due to non-determinacy of executing order of instructions.

Therefore, to preserve consistency of configurations, we need an extra synchronisation mechanism so that, during the traversing process performed by a thread, only one thread can access to the heap and connection map. The substitution could be realised by the following method [57]:

> First, we apply the translation $\lfloor\cdot\rfloor$ to a $\overset{ct}{\Longrightarrow}$-normal form configuration, and apply all Indirection rules to each equation $x = t$, thus bind all fragments of connections. If there is no active pair, then the execution process is recognised as finished. Otherwise, by using initial loading, we continue the execution for the new configuration.

We have to think, however, the cost for binding all fragments and the initial loading repeatedly.

### 3.3.2 amineLight

The amineLight [30] implementation was proposed in 2010, ten years after AMINE, and it is based on a textual calculus called the *lightweight calculus* (introduced in Section 4.2), which is a refined textual calculus introduced in Section 2.1.2.

Here, we review an abstract machine, called the *lightweight abstract machine*, based on the lightweight calculus. First, we define maps for connections between terms.

**Definition 3.3.11 (Operations for maps)**

*Let* P *be a set of pairs.*

- *We define a map* P *as a set of pairs:*

$$\mathrm{P}[n] \stackrel{\text{def}}{=} \begin{cases} m & ((n,m) \in \mathrm{P}) \\ \bot & (\textit{otherwise}) \end{cases}$$

- *We use the following notations to operate maps:*
  - $\mathrm{P}[n] := \bot$ *as the set* $(\mathrm{P} - \{(n,m)\})$ *for any* $m$,
  - $\mathrm{P}[n] := m$ *as the set* $(\mathrm{P}[n] := \bot) \cup \{(n,m)\})$.
- *We use the following notations to manage information of name connections:*
  - $\mathrm{P}[n \leftrightarrow m]$ *as a condition* $\mathrm{P}[n] = m$ *and* $\mathrm{P}[m] = n$,
  - $\mathrm{P} + (n \leftrightarrow m)$ *as the set* $((\mathrm{P}[n] := m)[m] := n$,
  - $\mathrm{P}[n \leftrightarrow m] := \bot$ *as the set* $((\mathrm{P}[n] := \bot)[m] := \bot$.

Next, we define a state of the abstract machine.

**Definition 3.3.12**

*A state of the lightweight abstract machine is defined by the following 5-tuple*

$$(\,\mathrm{E} \mid \mathrm{P} \mid \vec{t} \mid H \mid \Gamma\,)$$

*where*

- E *is an environment, which is a subset of* $\mathcal{N} \times \mathcal{T}$ *(where* $\mathcal{N}$ *is a set of names and* $\mathcal{T}$ *is the set of terms),*
- P *is a map for connections, which is a subset of* $\mathcal{N} \times \mathcal{N}$,
- $\vec{t}$ *is a sequence of terms,*
- $H$ *is a sequence of error equations that are not executable,*
- $\Gamma$ *is a sequences of equations. Equations are regarded as codes.*

*In those sequences, an empty sequence is denoted as "–".*

In contrast to the SECD machine [37], the stack $S$, the environment $E$ and the control $C$ in the SECD machine correspond to the term sequence $\vec{t}$, two maps E and P, and the equation sequence $\Gamma$ in this abstract machine respectively. There is no component which corresponds to the dump $D$ in the SECD machine because, during an execution of a rule, other rules are not called (application of a rule can be seen as an atomic operation).

We define operations Interaction and Error for equations such as $\alpha(\vec{t}) = \beta(\vec{s})$. Intuitively, Interaction is used to obtain an application result of "Interaction" rule in the lightweight calculus, and Error is used to classify equations where the "Interaction" rule cannot be applied.

**Definition 3.3.13**

*We define the maps* Interaction, Error *that take an equation which has agents at the root of the term on both sides, such as* $\alpha(\vec{t}) = \beta(\vec{s})$ *and return sequences of equations as follows:*

$$\mathsf{Interaction}(\alpha(\vec{t}) = \beta(\vec{s})) = \begin{cases} \Delta & (\textit{when } \langle \mid \alpha(\vec{t}) = \beta(\vec{s}) \rangle \rightarrow_{\text{int}} \langle \mid \Delta \rangle) \\ - & (\textit{otherwise}) \end{cases}$$

$$\mathsf{Error}(\alpha(\vec{t}) = \beta(\vec{s})) = \begin{cases} - & (\textit{when } \langle \mid \alpha(\vec{t}) = \beta(\vec{s}) \rangle \rightarrow_{\text{int}} \langle \mid \Delta \rangle) \\ \alpha(\vec{t}) = \beta(\vec{s}) & (\textit{otherwise}) \end{cases}$$

*where* $\rightarrow_{\text{int}}$ *is a reduction by "Interaction" rule in the lightweight calculus (introduced in Section 4.2).*

Figures 3.3, 3.4 and 3.5 give the semantics of the machine as a set of transitional rules of the form: $(\,\mathrm{E} \mid \mathrm{P} \mid \vec{t} \mid H \mid \Gamma\,) \Longrightarrow (\,\mathrm{E}' \mid \mathrm{P}' \mid \vec{t} \mid H' \mid \Gamma'\,)$. To aid readability we present the transitions in a table format. For example, the entry:

| | | Before | After |
|---|---|---|---|
| II.0 | Connections | $\mathrm{P}\ [x] = \bot$ | P |
| | Env. | $\mathrm{E}\ [x] = \bot$ | $\mathrm{E}\ [x] := \alpha(\vec{t})$ |
| | Code | $x = \alpha(\vec{t}),\ \Gamma$ | $\Gamma$ |

corresponds to:

$$(\,\mathrm{E}[x] = \bot \mid \mathrm{P}[x] = \bot \mid \vec{t} \mid H \mid x = \alpha(\vec{t}), \Gamma\,) \Longrightarrow (\,\mathrm{E}[x] := \alpha(\vec{t}) \mid \mathrm{P} \mid \vec{t} \mid H \mid \Gamma\,).$$

Intuitively, in Figures 3.4 and 3.5, each suffix *0*, *e* and *c* in rules means where names in the code are captured. For instance, the transition rule II.0, II.e and II.c for $x = t$ mean operations in the case that $x$ is not captured both in the environment and connections, that $x$ is in the environment map, and that $x$ is captured in the connection map respectively.

Next we define a compilation from a configuration to a machine state.

**Definition 3.3.14 (Compilation)**

*We define a translation* Compile *from a configuration into a machine state as follows:*

$$\mathsf{Compile}(\langle\ \vec{u} \mid \Delta\ \rangle) \stackrel{\text{def}}{=} (\,\emptyset \mid \emptyset \mid \vec{u} \mid - \mid \Gamma\,)$$

| | | Before | After |
|---|---|---|---|
| I | Error | $H$ | $\mathsf{Error}(\alpha(\vec{t}) = \beta(\vec{s})),\ H$ |
| | Code | $\alpha(\vec{t}) = \beta(\vec{s}),\ \Gamma$ | $\mathsf{Interaction}(\alpha(\vec{t}) = \beta(\vec{s})),\ \Gamma$ |

Figure 3.3: Transition for codes $\alpha(\vec{t}) = \beta(\vec{s})$

| | | Before | After |
|---|---|---|---|
| II.0 | Connections | $\mathrm{P}[x] = \bot$ | P |
| | Env. | $\mathrm{E}[x] = \bot$ | $\mathrm{E}[x] := \alpha(\vec{t})$ |
| | Code | $x = \alpha(\vec{t}),\ \Gamma$ | $\Gamma$ |
| II.e | Connections | $\mathrm{P}[x] = \bot$ | P |
| | Env. | $\mathrm{E}[x] = \beta(\vec{s})$ | $\mathrm{E}[x] := \bot$ |
| | Code | $x = \alpha(\vec{t}),\ \Gamma$ | $\beta(\vec{s}) = \alpha(\vec{t}),\ \Gamma$ |
| II.c | Connections | $\mathrm{P}[x \leftrightarrow y]$ | $\mathrm{P}[x \leftrightarrow y] := \bot$ |
| | Env. | $\mathrm{E}[x] = \bot, \mathrm{E}[y] = \bot$ | $\mathrm{E}[y] := \alpha(\vec{t})$ |
| | Code | $x = \alpha(\vec{t}),\ \Gamma$ | $\Gamma$ |
| II.– | Code | $\alpha(\vec{t}) = x,\ \Gamma$ | $x = \alpha(\vec{t}),\ \Gamma$ |

Figure 3.4: Transitions for codes $x = \alpha(\vec{t})$ and $\alpha(\vec{t}) = x$

*where $\Gamma$ is a sequence of equations that is the result of fixing an order of the multiset of equations $\Delta$, where the order may be decided arbitrarily. We use $\Gamma$ to range over sequences of equations.*

We obtain the execution result by using the following `update` translation:

**Definition 3.3.15**

*We define the operation* `update` *as follows:*

- $\texttt{update}(\,\mathrm{E} \mid \mathrm{P}[x \leftrightarrow y] \mid \vec{t} \mid H \mid -\,) = \texttt{update}(\,\mathrm{E} \mid \mathrm{P} \mid \vec{t}[x/y] \mid H \mid -\,)$,
- $\texttt{update}(\,\mathrm{E}[x \mapsto s] \mid [] \mid \vec{t} \mid \Theta \mid -\,) = \texttt{update}(\,\mathrm{E}[s/x] \mid [] \mid \vec{t}[s/x] \mid H \mid -\,)$,
- $\texttt{update}(\,[] \mid [] \mid \vec{t} \mid H \mid -\,) = \vec{t}$.

**Example 3.3.16**

*We show the computation of* $\langle\ r \mid \mathtt{Add}(r, \mathtt{Z}) = \mathtt{S}(\mathtt{Z})\ \rangle$. *We start the abstract machine from the following state:*

$$(\,[] \mid [] \mid r \mid - \mid \mathtt{Add}(r, \mathtt{Z}) = \mathtt{S}(\mathtt{Z})\,)$$

$(\,[] \mid [] \mid r \mid - \mid \mathtt{Add}(r, \mathtt{Z}) = \mathtt{S}(\mathtt{Z})\,)$

| | | Before | After |
|---|---|---|---|
| III.0_0 | Connections | $\mathrm{P}[x] = \bot, \mathrm{P}[y] = \bot$ | $\mathrm{P} + (x \leftrightarrow y)$ |
| | Env. | $\mathrm{E}[x] = \bot, \mathrm{E}[y] = \bot$ | E |
| | Code | $x = y,\ \Gamma$ | $\Gamma$ |
| III.0_e | Connections | $\mathrm{P}[x] = \bot, \mathrm{P}[y] = \bot$ | P |
| | Env. | $\mathrm{E}[x] = \bot, \mathrm{E}[y] = \alpha(\vec{t})$ | $(\mathrm{E}[x] := \alpha(\vec{t}))[y] := \bot$ |
| | Code | $x = y,\ \Gamma$ | $\Gamma$ |
| III.0_c | Connections | $\mathrm{P}[x] = \bot, \mathrm{P}[y \leftrightarrow w]$ | $(\mathrm{P}[y \leftrightarrow w] := \bot) + (x \leftrightarrow w)$ |
| | Env. | $\mathrm{E}[x] = \bot, \mathrm{E}[y] = \bot$ | E |
| | Code | $x = y,\ \Gamma$ | $\Gamma$ |
| III.e_0 | Connections | $\mathrm{P}[x \leftrightarrow y] = \bot$ | P |
| | Env. | $\mathrm{E}[x] = \alpha(\vec{t}), \mathrm{E}[y] = \bot$ | $(\mathrm{E}[x] := \bot)[y] := \alpha(\vec{t})$ |
| | Code | $x = y,\ \Gamma$ | $\Gamma$ |
| III.e_e | Connections | $\mathrm{P}[x] = \bot, \mathrm{P}[y] = \bot$ | P |
| | Env. | $\mathrm{E}[x] = \alpha(\vec{t}), \mathrm{E}[y] = \beta(\vec{s})$ | $(\mathrm{E}[x] := \bot])[y] := \bot$ |
| | Code | $x = y,\ \Gamma$ | $\alpha(\vec{t}) = \beta(\vec{s}),\ \Gamma$ |
| III.e_c | Connections | $\mathrm{P}[x] = \bot, \mathrm{P}[y \leftrightarrow w]$ | $\mathrm{P}[y \leftrightarrow w] := \bot$ |
| | Env. | $\mathrm{E}[x] = \alpha(\vec{t}), \mathrm{E}[y] = \bot$ | $(\mathrm{E}[x] := \bot)[w] := \alpha(\vec{t})$ |
| | Code | $x = y,\ \Gamma$ | $\Gamma$ |
| III.c_0 | Connections | $\mathrm{P}[x \leftrightarrow z], \mathrm{P}[y] = \bot$ | $(\mathrm{P}[x \leftrightarrow z] := \bot) + (y \leftrightarrow z)$ |
| | Env. | $\mathrm{E}[x] = \bot, \mathrm{E}[y] = \bot$ | E |
| | Code | $x = y,\ \Gamma$ | $\Gamma$ |
| III.c_e | Connections | $\mathrm{P}[x \leftrightarrow z], \mathrm{P}[y] = \bot$ | $\mathrm{P}[x \leftrightarrow z] := \bot$ |
| | Env. | $\mathrm{E}[x] = \bot, \mathrm{E}[y] = \alpha(\vec{t})$ | $(\mathrm{E}[y] := \bot)[z] := \alpha(\vec{t})$ |
| | Code | $x = y,\ \Gamma$ | $\Gamma$ |
| III.c_c | Connections | $\mathrm{P}[x \leftrightarrow z], \mathrm{P}[y \leftrightarrow w]$ | $((\mathrm{P}[x \leftrightarrow z] := \bot)[y \leftrightarrow w] := \bot) + (z \leftrightarrow w)$ |
| | Env. | $\mathrm{E}[x] = \bot, \mathrm{E}[y] = \bot$ | E |
| | Code | $x = y,\ \Gamma$ | $\Gamma$ |

Figure 3.5: Transitions for codes $x = y$

$$\Longrightarrow (\ [] \mid [] \mid r \mid - \mid r = \mathtt{S}(x), \mathtt{Z} = \mathtt{Add}(x, \mathtt{Z})\ ) \tag{I}$$

$$\Longrightarrow (\ [r \mapsto \mathtt{S}(x)] \mid [] \mid r \mid - \mid \mathtt{Z} = \mathtt{Add}(x, \mathtt{Z})\ ) \tag{II.0}$$

$$\Longrightarrow (\ [r \mapsto \mathtt{S}(x)] \mid [] \mid r \mid - \mid x = \mathtt{Z})\ ) \tag{I}$$

$$\Longrightarrow (\ [r \mapsto \mathtt{S}(x)][x \mapsto \mathtt{Z}] \mid [] \mid r \mid - \mid -\ ). \tag{II.0}$$

$$\mathtt{update}(\ [r \mapsto \mathtt{S}(x)][x \mapsto \mathtt{Z}] \mid [] \mid r \mid - \mid -\ )$$

$$= \mathtt{update}(\ [r \mapsto \mathtt{S}(\mathtt{Z})] \mid [] \mid r \mid - \mid -\ ) = \mathtt{S}(\mathtt{Z}).$$

## 3.4 Comparison of encoding methods

In this section, we compare methods of encoding nets among evaluators which we have discussed in the previous section so that we can compare them in terms of efficiency. To unify the data-structures, we use a standardised implementation model such that, instead of indexes of arrays, pointers are used for entries of the memory heaps.

A net configuration is represented by the following data (which is a similar configuration implemented by INET discussed in Section 3.2.1):

- a heap of agent nodes $\Gamma$,
- a stack of active pairs $AP$,
- an array of the interface $I$,
- a rule table $R$,
- a runtime environment.

This is summarised by Figure 3.6.

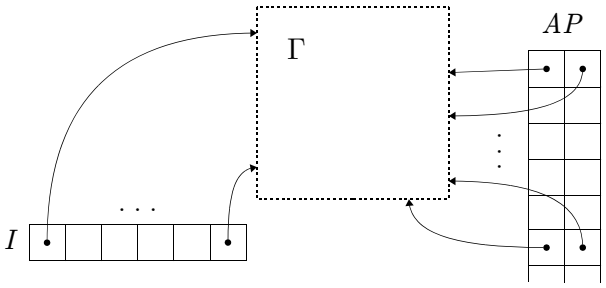



Figure 3.6: Configuration

### 3.4.1 Undirected graph encoding

We call the encoding method of $\mathsf{in}^2$, PIN and INET *Undirected graph encoding*. The agent node is represented using the following C codes:

```
typedef struct Agent {
  int id;
  struct Port *port[MAX_PORT];
} Agent;
```

```
typedef struct Port {
  Agent *agent;
  int portNum;
} Port;
```

For simplicity, we fix the size of ports with a pre-defined constant `MAX_PORT`. The `Port` has two components: a pointer to a connected agent node and a port number of the agent node where 0 is used for the principal ports and $n > 0$ is for an auxiliary port. We draw, when `MAX_PORT` is 2, agents $\alpha$, $\beta$ whose arities are 2 and 1 respectively as follows:


Similar to the INET mode, an interface is represented as a graph node whose id is `ID_NAME` and arity is 1. Agents and interface nodes are allocated and de-allocated using the following functions:

```
/* to allocate and de-allocate agents and the interface */
Agent *mkAgent(int id);
Agent *mkInterface();
void freeAgent(Agent *a);
```

We have two alternative methods for port connections: one is the method used in INET (and PIN), and the other method is used in in$^2$. As an example, the net `S(Z)` is represented in INET using the graph shown on the left and the same term is represented in in$^2$ using the graph shown on the right:

(a) the INET method (b) the in$^2$ method


To capture both representations in our C code, we introduce a constant `IN2` which we can use to switch between the INET and in$^2$ representations:

```
#define connect(a1, p1, a2, p2) { \
#ifndef IN2 \
  a1->port[p1]->agent   = a2; \
  a1->port[p1]->portNum = p2; \
  a2->port[p2]->agent   = a1; \
```

```
  a2->port[p2]->portNum = p1; \
  if (p1==0 && p2==0) pushActive(a1, a2)
#else
  if (p1 == 0 && p2 == 0) { \
    pushActive(a1, a2); \
  } else if (p1 == 0 && p2 != 0) { \
    a2->port[p2]->agent   = a1; \
    a2->port[p2]->portNum = p1; \
  } else if (p1 != 0 && p2 == 0) { \
    a1->port[p1]->agent   = a2; \
    a1->port[p1]->portNum = p2; \
  } else { \
    a1->port[p1]->agent   = a2; \
    a1->port[p1]->portNum = p2; \
    a2->port[p2]->agent   = a1; \
    a2->port[p2]->portNum = p1; \
  }
#endif
#define getPort(a, p) (a->port[p])
```

Active pairs are managed by the following LIFO stack:

```
typedef struct Active {
  Agent *a1;
  Agent *a2;
} Active;
Active ActivePairs[MAX_ACTIVE];

/* to manipulate ActivePairs */
int Ptr_APS = -1; // index of the stack of equations
void pushActive(Agent *a1, Agent *a2) {
  Ptr_APS++;
  if (Ptr_APS >= MAX_ACTIVE) {
    puts("ERROR"); exit(-1);
  }
  ActivePairs[Ptr_APS].a1 = a1;
```

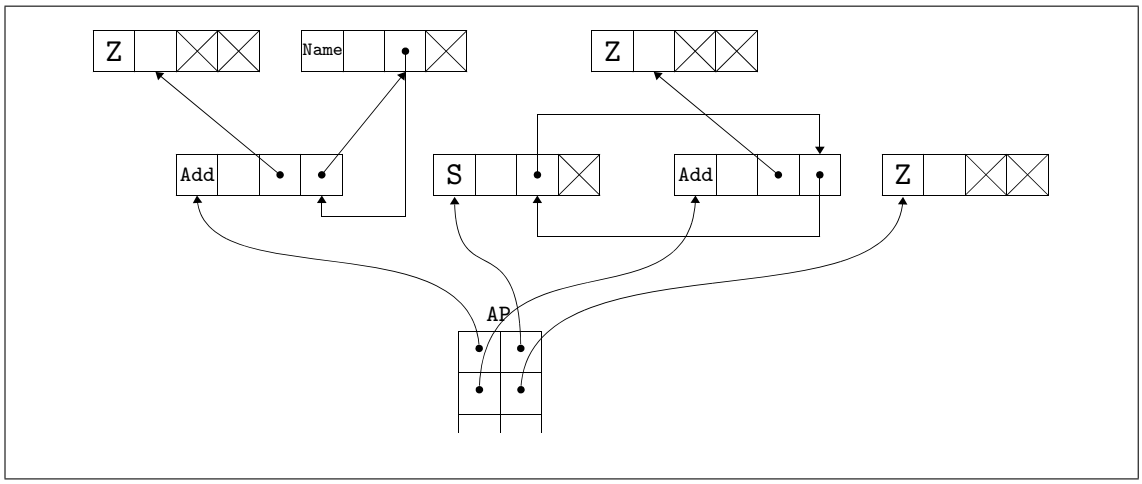
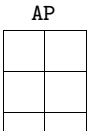



Figure 3.7: Undirected method: the net in Figure 2.2

```
  ActivePairs[Ptr_APS].a2 = a2;
}
int PopActivePair(Agent **a1, Agent **a2) {
  if (Ptr_APS >= 0) {
    *a1 = ActivePair[Ptr_APS].a1;
    *a2 = ActivePair[Ptr_APS].a2;
    Ptr_APS--;
    return 1;
  }
  return 0;
}
```

The active pair stack is drawn as follows:




For instance, the net in Figure 2.2 is drawn in the $\mathsf{in}^2$ method as shown in Figure 3.7.

Runtime functions are defined as follows:

```
void eval();
void putsAgent(Agent *a);
```

where

- the function `eval` operates all of stacked active pairs until the stack becomes empty as follows:

```
void eval() {
  Agent *a1, *a2;
  while (popActive(&a1, &a2)) {
    R[a1][a2](a1, a2);
  }
}
```

- the function `putsAgent` takes a pointer to an entry of `Heap` and outputs the image as strings.

For instance, a function for the interaction rule $\texttt{Add}(x, x) \bowtie \texttt{Z}$ and $\texttt{Add}(y, \texttt{S}(w)) \bowtie \texttt{S}(\texttt{Add}(y, w))$ are written as the following functions:

```
void Add_Z(Agent *a1, Agent *a2) {
  connect(getPort(a1,1).agent, getPort(a1,1).portNum,
          getPort(a1,2).agent, getPort(a1,2).portNum);
  freeAgent(a1);
  freeAgent(a2);
}

void Add_S(Agent *a1, Agent *a2) {
  Agent *newS = mkAgent(ID_S);
  Agent *newAdd = mkAgent(ID_Add);
  connect(newS, 1, newAdd, 2);
  connect(getPort(a1,1).agent, getPort(a1,1).portNum,
          newAdd, 1);
  connect(getPort(a1,2).agent, getPort(a1,2).portNum,
          newS, 0);
  connect(getPort(a2,1).agent, getPort(a2,1).portNum,
          newAdd, 0);
  freeAgent(a1);
  freeAgent(a2);
}
```

and the net in Figure 2.1 is evaluated by these functions as shown in Figure 3.8.

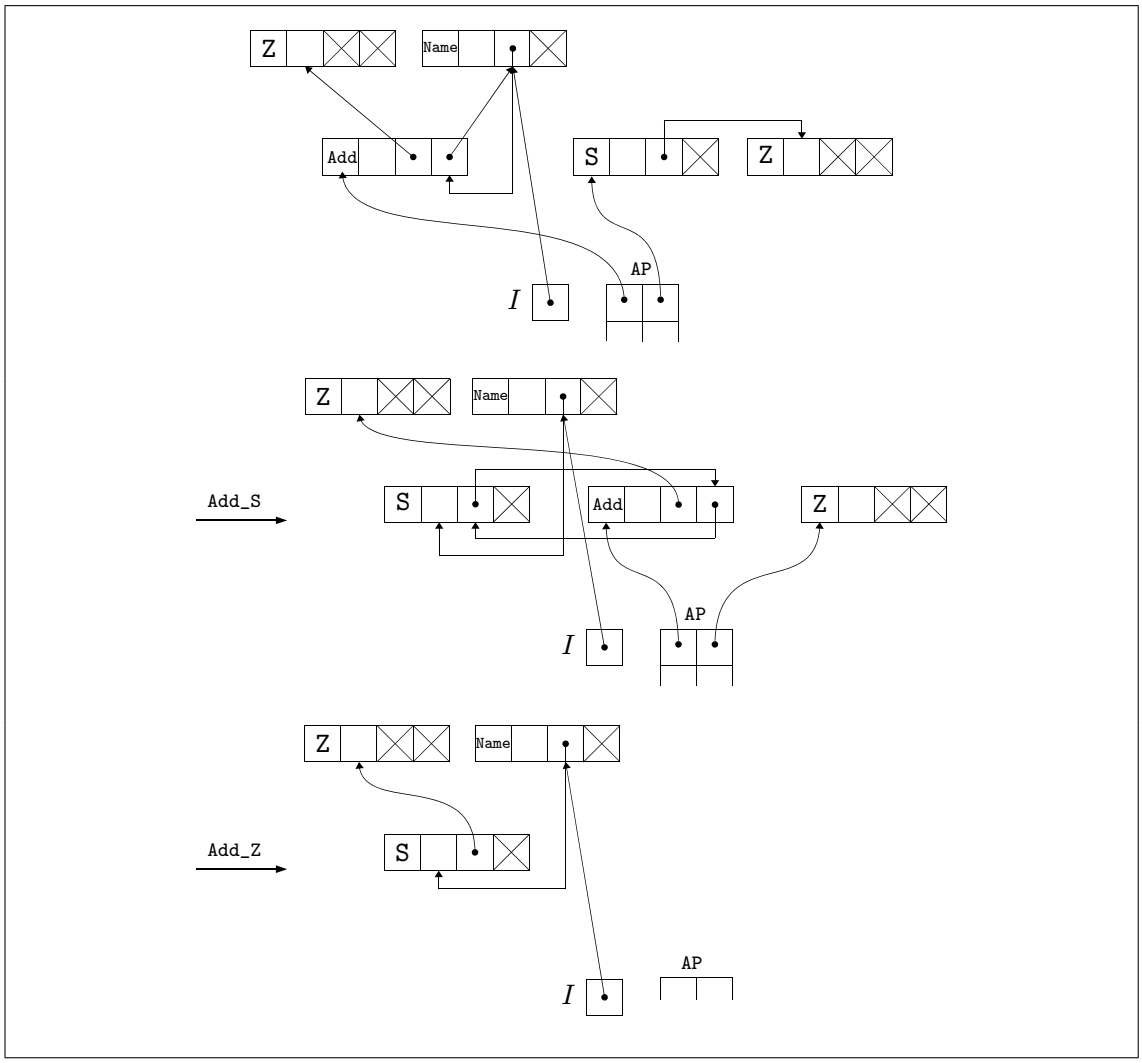


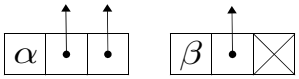

Figure 3.8: Undirected encoding method: evaluation of the net in Figure 2.1

### 3.4.2 Directed graph encoding

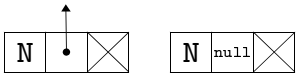
We call the encoding method of amineLight *Directed graph encoding*. The agent node is represented using the following C codes:

```
typedef struct Agent {
  int id;
  struct Agent *port[MAX_PORT];
} Agent;
```

In an agent node, a `port` corresponds to an auxiliary port and stores the information where the auxiliary port is connected to. A pointer to an agent node is regarded as the principal port of the agent. We draw, when `MAX_PORT` is 2, agents $\alpha$, $\beta$ whose arities are 2 and 1 respectively as follows:



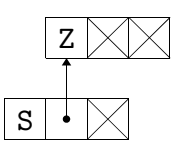

In contrast to the Undirected graph encoding method, the textual calculus requires names as terms. In this method, names are represented as agent nodes whose id is 0 denoted as `ID_NAME` and arity is 1. The port is filled with a `null` pointer when it has no connection, as shown below:



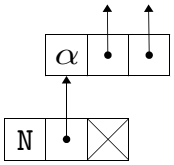

Agent and name nodes are allocated and de-allocated using the following functions:

```
/* to allocate and de-allocate names and agents */
Agent *mkAgent(int id);
Agent *mkName();
void freeAgent(Agent *a);
```

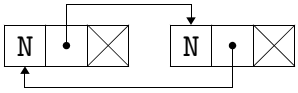
A connection between a principal port and an auxiliary port, i.e. an agent node `a` and an auxiliary port `b->port[`$n$`]` is represented as an assignment: `b->port[`$n$`] = a`. For instance, the net `S(Z)` is represented as `aS->port[0]=aZ` where `aS` and `aZ` are variables referring to the nodes corresponding to agents `S` and `Z` respectively. This net is drawn as follows:



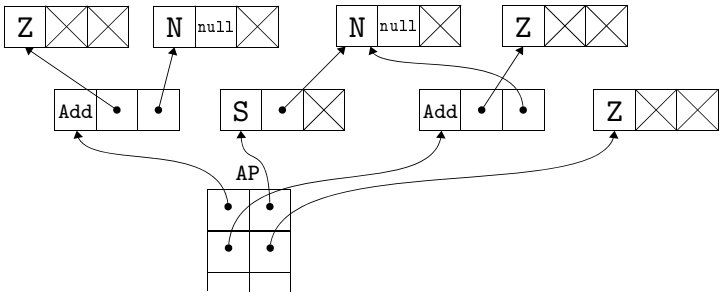
The abstract machine of amineLight has two components to manage connections: an environment E and a map for connections P. According to the source file that was obtained by the link in the paper [30], E$[x] = t$ is represented as `x->port[0]=t`. It is drawn as follows:




With respect to a connection P$[x \leftrightarrow y]$, it is presented by two connections: `x->port[0]=y` and `y->port[0]=x`. It is drawn as follows:




In the textual calculus, active pairs are represented by equations. Graphically, these equations are represented as two pointers to agent nodes. Thus, in order to manage equations we deploy the same functions used in Undirected encoding method. The net in Figure 2.2 is drawn in Directed encoding method as follows:




Interaction rules are represented as functions that takes two pointers of equations and make equations according to the rules. For instance, rules in Example 4.2.5 are written as follows:

```
void Add_Z(Agent *a1, Agent *a2) {
  /* x1=x2 */
  pushActive(a1->port[0], a1->port[1]);

  freeAgent(a1);
  freeAgent(a2);
}
```

```
void Add_S(Agent *a1, Agent *a2) {
  /* Add(x1,w)=y */
  Agent *w = mkName();
  Agent *aAdd = mkAgent(ID_Add);
  aAdd[0] = a1->port[0];
  aAdd[1] = w;
  pushActive(aAdd, a2->port[0]);

  /* x2=S(w) */
  Agent *aS = mkAgent(ID_S);
  aS[0] = w;
  pushActive(a1->port[1], aS);

  freeAgent(a1);
  freeAgent(a2);
}
```

Besides interaction rules, there are transitions for equations contain names as shown in Figure 3.4 and 3.5. Thus, the runtime function `eval` is written as follows:

```
void eval() {
  Agent *a1, *a2;
  while (popActive(&a1, &a2)) {
    if (a2->id != ID_NAME) {
      if (a1->id != ID_NAME) {
        R[a1->id][a2->id](a1, a2);
      } else {
        /* operations for x=Alpha(x1,...,xn) */
          ⋮
      }
    } else {
      /* operations for Alpha(x1,...,xn)=y and x=y */
        ⋮
    }
  }
}
```

Transition rules II.0, II.c and II.e in Figure 3.4 are written as follows and these are placed under the comment `/* operations for x=Alpha(x1,...,xn) */`:

```
// a1 is a name, a2 is an agent
if (a1->port[0] == NULL) {
  /* II.0 */
  a1->port[0] = a2;
} else if ((a1->port[0])->id != ID_NAME) {
  /* II.e */
  Agent *a1p0 = a1->port[0];
  freeAgent(a1);
  a1=a1p0;
  pushActive(a1,a2);
} else {
  /* II.c */
  (a1->port[0])->port[0] = a2;
  freeAgent(a1);
}
```

These operations are drawn as Figure 3.9. For transition rules in Figure 3.5, see Appendix A.1.

The net in Figure 2.1 is evaluated by these functions as shown in Figure 3.10.

### 3.4.3 Experimental results

In this section we compare those encoding methods in terms of execution time by using the following three benchmark programs:

- Fibonacci number $F_n$ defined in Figure 2.4,
- Ackermann function $A$ defined in Figure 2.5,
- Application of Church numerals $n = \lambda f.\lambda x.f^n x$ and $I = \lambda x.x$. The encoding method of those lambda terms is described in the paper [43].

The execution was performed on a laptop PC consisting of an Intel Core-i7(2.4GHz) and 16GB RAM using Ubuntu 12.04 LTS operating system. The execution time was measured using the UNIX `time` command. The Table 3.1 shows execution time in seconds of those benchmark programs by using INET, $\mathsf{in}^2$ and amineLight (denoted as "Light")

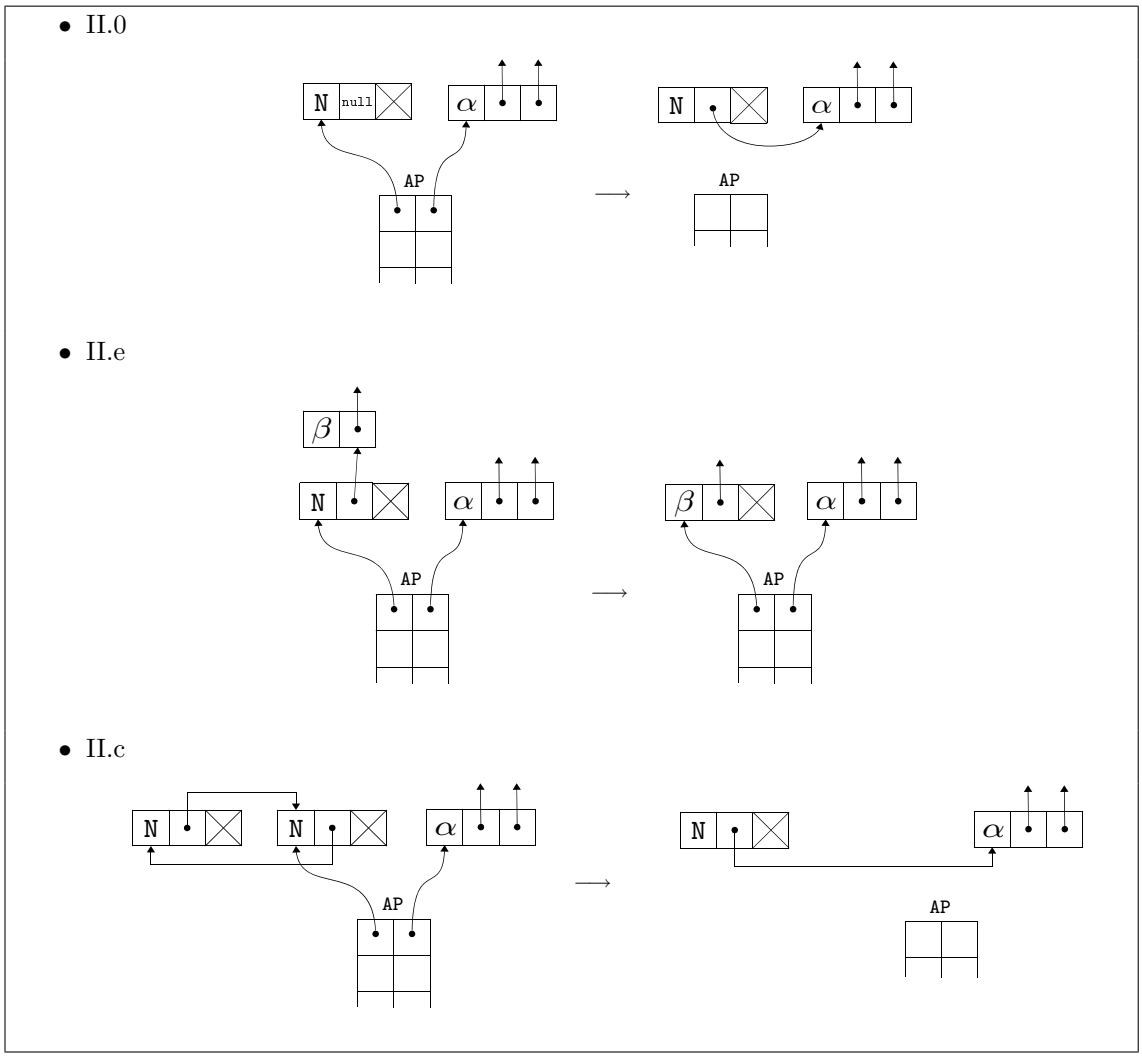



Figure 3.9: Transition rules in Figure 3.4

encoding methods. Those are compiled by gcc 4.6.3 with `-O3` optimisation option. Generally, the encoding method of $\mathsf{in}^2$ is more efficient than INET because the port connections in $\mathsf{in}^2$ are simpler. The amineLight encoding is also simpler than INET, and amineLight is faster except for the case in the execution of Application of Church numerals. It requires a lot of operations for names as shown in Table 3.2 where the items "Interactions" and "Names" mean the numbers of operations for interactions and names respectively. According to the increasing number of name operations, the execution time also increases, and thus operations for those cause less efficiency.

In terms of the cost, therefore, the Undirected encoding method of $\mathsf{in}^2$ is the best. In comparison with Directed encoding method, it is faster about from 10% to 70%. On the other hand, the advantage of Directed encoding is parallelisation, and it is important to bear in mind that this could improve performance. Further discussion of this topic with our new method will be held in Section 4.5.3.

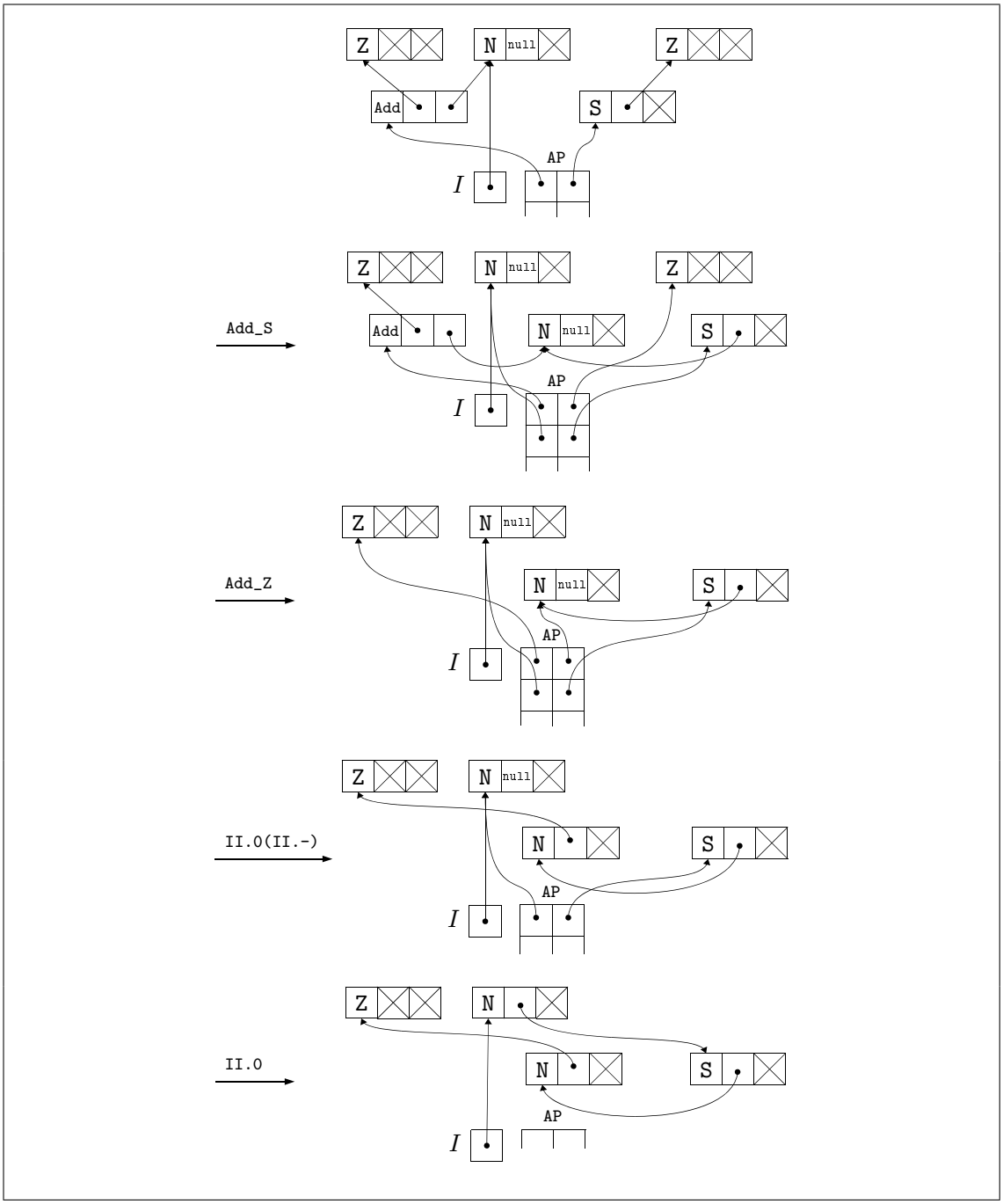



Figure 3.10: Directed encoding method: evaluation of the net in Figure 2.1

| | Undirected(INET) | Undirected($\mathsf{in}^2$) | Directed(Light) | Light/$\mathsf{in}^2$ |
|---|---|---|---|---|
| $F_{32}$ | 1.58 | 1.37 | 1.52 | 1.11 |
| $F_{33}$ | 2.62 | 2.29 | 2.52 | 1.10 |
| $F_{34}$ | 4.37 | 3.80 | 4.21 | 1.11 |
| $A(3,10)$ | 1.77 | 1.42 | 1.59 | 1.12 |
| $A(3,11)$ | 7.12 | 5.73 | 6.44 | 1.13 |
| $A(3,12)$ | 29.47 | 24.01 | 26.39 | 1.13 |
| `2 7 6 I I` | 0.73 | 0.71 | 1.26 | 1.77 |
| `2 7 7 I I` | 2.12 | 2.13 | 3.58 | 1.68 |

Table 3.1: The execution time in seconds on the standardised implementation model

| | Interactions | Names | Names/Interactions |
|---|---|---|---|
| $F_{32}$ | 74636718 | 51008017 | 0.68 |
| $F_{33}$ | 123315177 | 82532797 | 0.67 |
| $F_{34}$ | 203654818 | 133540964 | 0.66 |
| $A(3,10)$ | 134103148 | 134094952 | 1.00 |
| $A(3,11)$ | 536641652 | 536625264 | 1.00 |
| $A(3,12)$ | 2147025020 | 2146992248 | 1.00 |
| `2 7 6 I I` | 15676873 | 43111255 | 2.75 |
| `2 7 7 I I` | 46118916 | 126826871 | 2.75 |

Table 3.2: The number of operations in Directed encoding method

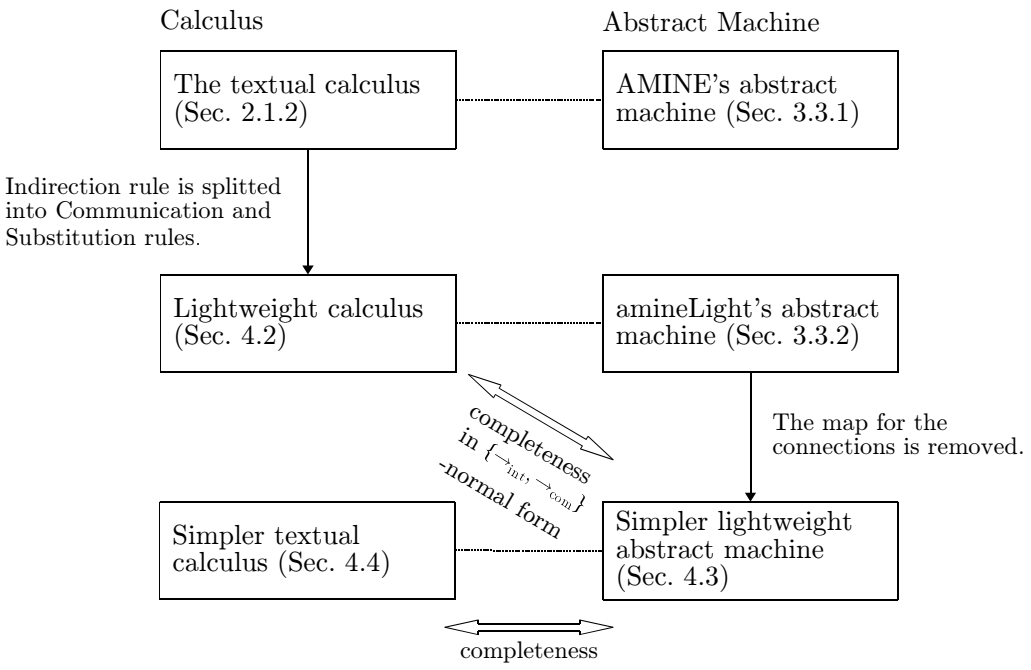
## 3.5 Summary

In this chapter we have reviewed evaluators of interaction nets, where the focus is on efficiency. The efficiency of execution can depend on the skill of the developer(s), and thus it is important to examine properties of these evaluators in a uniform way. For this purpose we introduced a standardised implementation model, explicating internal data-structures in those evaluators. This model uses, instead of indexes of arrays, pointers for entries of the memory heaps, and builds nets according to the methods of encoding nets in those evaluators. Of course, there is scope in the model for a number of design choices and various models known in the literature, and thus the comparison result that we have shown is just a criteria in the standardised model. However, it is useful to reason about properties of those various models in a uniform way. In the next chapter, we introduce our new method based on the standardised model, and discuss efficiency and suitability for parallel execution.

# Chapter 4

# Single link encoding method

In this chapter we propose a new method for implementing interaction nets. Our new method is a refinement of the method used in amineLight in that we use only single links to encode nets as a tree-like data-structure. First, we explain why this method is required. Next, we introduce the new abstract machine and textual calculus, which are called a simpler lightweight abstract machine and a simpler textual calculus, as a refinement of amineLight's ones. An overview of the relationship between these calculi and abstract machines is illustrated in the diagram below. We also give an encoding method based on the standardised implementation model, and finally we give a comparison with other encoding methods in terms of runtime efficiency.

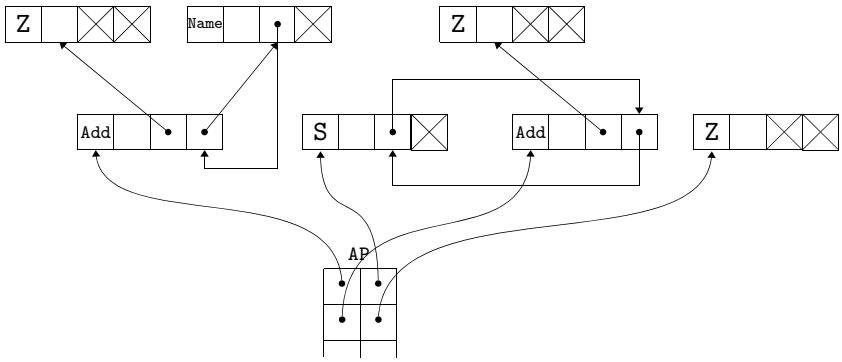


## 4.1 Motivation

An important property of interaction nets is locality since all rewritings are performed locally, and we'll see later in Chapter 7 that interaction nets are very well suited for parallel implementations. In terms of the locality, we review encoding methods in INET, in$^2$ and amineLight, which we have discussed in Section 3.4.

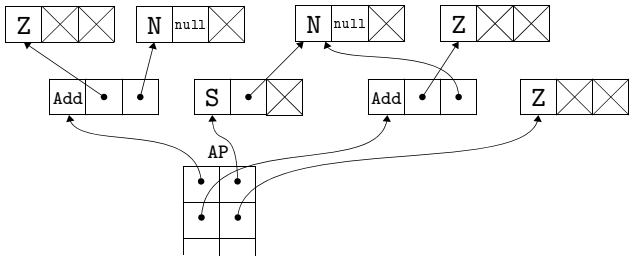
Generally, when an active pair is reduced, a new net is created according to an interaction rule. The interface of the right hand side net of the rule must be connected to ports that were connected to the active pair. Thus, two active pairs that are not connected to each other via auxiliary ports can be reduced simultaneously. Reduction of two active pairs that are connected via an auxiliary port(s) of an interacting agent need to be managed differently because each rewrite will update the same set of auxiliary ports.

A net in Figure 2.2 is an example of this situation, which is also introduced as an example to consider the cost of parallel execution in the paper [49]. The two active pairs are connected to each other via the auxiliary port of the interacting agent `S` and `Add`, and this connection information must be preserved when the active pairs are reduced at once. In an execution model of in$^2$ discussed in Section 3.4, which is considered as a simpler version of INET, this net is represented as follows:

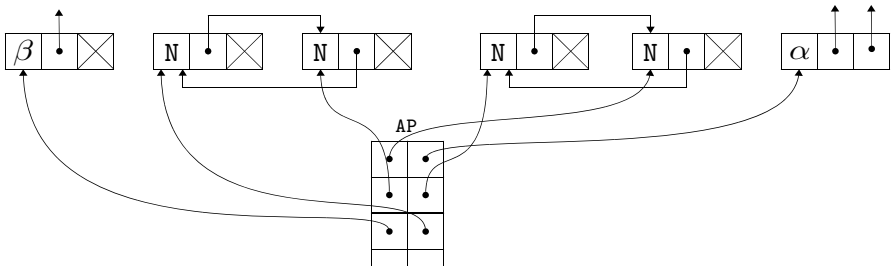



An interaction involves manipulation of pointers stored in auxiliary ports by the operation `connect`. For instance, at the second step `Add_Z` in Figure 3.8 two pointers which are stored in the auxiliary ports of `Add` are manipulated, and `S(Z)` is created as the manipulation result. When stacked active pairs are reduced in parallel, we have to check whether the active pairs are connected via their auxiliary ports, and lock those connected active pairs so that the ports can be preserved until the rewriting finishes. In this net, the second active pair in the stack should be locked while the top one is being reduced. This checking process could be spread into other parts of the net globally.

The following is an example of the net in an execution model of amineLight:




The connection via their auxiliary ports is preserved by a name, and thus reduction of the two active pairs are performed in parallel as long as the operation of the name is managed as a critical section. However, the connections between names are represented as mutual links and we need to check for the lock and this can also spread globally. Take the following as an example:




The three elements of the stack should not be performed at once, so the checking process is required.

The mutual links affect the locality and thus we propose a new method of encoding so that a connection between names can be represented by a single link.

## 4.2 Lightweight textual calculus

A textual calculus is introduced in Section 2.1.2, and it provides a simple and dynamic semantics for interaction nets. There are, however, two concerns:

- One is that the calculus needs extra rewriting steps to reduce a given net to a normal form. For instance, the net in Figure 2.1 is reduced by two steps in the graphical calculus while the same net is reduced by six steps in the textual calculus as shown in Example 2.1.9.

- The other is that it does not have the strong confluence property, which is an important property, while it holds in the graphical calculus. This is because the Indirection rule can allow two calculation results. For instance, a configuration $\langle\ \vec{t}\ \mid\ \alpha(x) = y, \beta(y) = x\ \rangle$ can be reduced to both $\langle\ \vec{t}\ \mid\ \alpha(\beta(y)) = y\ \rangle$ and

$\langle\ \vec{t}\ \mid\ \beta(\alpha(x)) = x\ \rangle$ by the Indirection rule, and there is no confluence for these configurations.

To solve these problems, a lightweight calculus was proposed, and the amineLight implementation is based on this calculus.

In this section we introduce the lightweight calculus and show properties of the calculus.

### 4.2.1 Lightweight interaction rules

The notation of Lafont's style generates (redundant) equations which will be reduced by the Indirection rule. In particular, if an auxiliary port of an active pair in a rule is connected to another auxiliary port, the application of "Interaction" rule will generate an equation with a variable $x$ on one side of the equation. Since all variables appear twice in a rule, $x$ will eventually be eliminated using the Indirection rule. For example, this can be traced in Example 2.1.9 where the equation $\mathtt{Z} = y'$ is generated in the configuration after applying the first rule $\mathtt{Add}(y, \mathtt{S}(w)) \bowtie \mathtt{S}(\mathtt{Add}(y, w))$. In other words, the application of "Interaction" rule to an active pair $(\alpha, \beta)$ where $\alpha(\vec{t_1}, x, \vec{t_2}) \bowtie \beta(\vec{s_1}) \in \mathcal{R}$ will generate a configuration where the Indirection rule is applicable.

In order to eliminate the generation of redundant equations, we introduce an alternative notation to represent interaction rules. We represent rules using the following syntax:

$$lhs \Rightarrow rhs$$

where *lhs* consists of an equation between the two interacting agents and *rhs* is a list of equations which represent the right-hand side net. All rules $\alpha(\vec{t}) \bowtie \beta(\vec{s})$ in Lafont's style can be written using our notation:

$$\alpha(\vec{x}) = \beta(\vec{y}) \Rightarrow \vec{x} = \vec{t}, \vec{y} = \vec{s} \qquad \text{where } \vec{x}, \vec{y} \text{ are meta-variables for terms.}$$

As a concrete example, the rule $\mathtt{Add}(\mathtt{S}(x), y) \bowtie \mathtt{S}(\mathtt{Add}(x, y))$ can be represented as

$$\mathtt{Add}(a_1, a_2) = \mathtt{S}(b_1) \Rightarrow a_1 = \mathtt{S}(x), a_2 = y, b_1 = \mathtt{Add}(x, y).$$

Moreover we can simplify rules by replacing equals for equals. The above rule can be simplified to:

$$\mathtt{Add}(a_1, a_2) = \mathtt{S}(b_1) \Rightarrow a_1 = \mathtt{S}(x), b_1 = \mathtt{Add}(x, a_2).$$

We obtain, therefore, a more efficient computation by using the notation of term rewriting systems.

**Definition 4.2.1 (Lightweight interaction rules)**

*A lightweight rule* $r \in \mathcal{R}_{\text{lt}}$ *has the form:*

$$\alpha(x_1, \ldots, x_n) = \beta(y_1, \ldots, y_k) \Rightarrow \Delta$$

*where* $\alpha, \beta \in \Sigma$, $\text{ar}(\alpha) = n, \text{ar}(\beta) = k$, *and* $x_1, \ldots, x_n, y_1, \ldots, y_k$ *are meta-variables for terms. Each meta-variable occurs exactly twice in a rule: once on the left hand side (LHS) and once on the right hand side (RHS). The set* $\mathcal{R}_{\text{lt}}$ *contains at most one rule between any pair of agents;* $\mathcal{R}_{\text{lt}}$ *is closed under symmetry — if* $\alpha(\vec{x}) = \beta(\vec{y}) \Rightarrow \Delta \in \mathcal{R}_{\text{lt}}$ *then* $\beta(\vec{y}) = \alpha(\vec{x}) \Rightarrow \Delta \in \mathcal{R}_{\text{lt}}$.

**Definition 4.2.2 (Instance of a rule)**

*If* $r$ *is a lightweight rule* $\alpha(x_1, \ldots, x_n) = \beta(y_1, \ldots, y_m) \Rightarrow \Delta$, *then* $\widehat{\Delta}$ *denotes a new generic* instance *of* $r$, *that is, a copy of* $\Delta$ *where we introduce a new set of bound names so that those new names do not overlap with others already exist, but leave the free names (parameters) unchanged. Example: if* $\Delta$ *is* $\alpha(x, x) = \beta(a)$, *then* $\widehat{\Delta}$ *is* $\alpha(y, y) = \beta(a)$, *where* $y$ *is a fresh name.*

### 4.2.2 Decomposing Indirection rule

Let us now examine the Indirection rule of the calculus which eliminates bound variables by means of variable substitution. The application of this rule will search through the list of terms to locate a term which contains an occurrence of a particular variable. In order to reduce the searching costs, Pinto's abstract machine [57], which is based on the textual calculus introduced in Section 2.1.2, attaches a list of variables to the head of every term. This again introduces management overheads, hence the increase in the number of operations required to perform rewriting.

Taking into consideration that every change of connection does not affect interactions directly, it turns out that we do not have to perform all substitutions eagerly. Therefore we decompose the Indirection rule into: *Communication rule* that will replace just a name, and *Substitution rule* that will perform other substitutions.

**Definition 4.2.3 (Communication and Substitution rules)**

*We define Communication and Substitution rules as follows:*

**Communication:**

$$\langle\ \vec{u}\ \mid\ x = t, x = s, \Delta\ \rangle \rightarrow_{\text{com}} \langle\ \vec{u}\ \mid\ t = s, \Delta\ \rangle.$$

**Substitution:**

$$\langle\ \vec{u}\ \mid\ x = s, \beta(\vec{t}) = u, \Delta\ \rangle \rightarrow_{\text{sub}} \langle\ \vec{u}\ \mid\ \beta(\vec{t})[s/x] = u, \Delta\ \rangle \quad \textit{where } \beta \in \Sigma \textit{ and } x \textit{ occurs}$$

*in* $\vec{t}$.

### 4.2.3 Lightweight calculus

Here, we introduce the lightweight calculus by using Definitions 4.2.1, 4.2.2 and 4.2.3 with the Collect rule in the original textual calculus in Section 2.1.2:

**Definition 4.2.4 (Lightweight reduction rules)**

*We define Lightweight reduction rules as follows:*

**Communication:**

$\langle\ \vec{u}\ \mid\ x = t, x = s, \Delta\ \rangle \rightarrow_{\mathrm{com}} \langle\ \vec{u}\ \mid\ t = s, \Delta\ \rangle$.

**Substitution:**

$\langle\ \vec{u}\ \mid\ x = s, \beta(\vec{t}) = u, \Delta\ \rangle \rightarrow_{\mathrm{sub}} \langle\ \vec{u}\ \mid\ \beta(\vec{t})[s/x] = u, \Delta\ \rangle$ *where* $\beta \in \Sigma$ *and* $x$ *occurs in* $\vec{t}$.

**Collect:**

$\langle\ \vec{u}\ \mid\ x = s, \Delta\ \rangle \rightarrow_{\mathrm{col}} \langle\ \vec{u}[s/x]\ \mid\ \Delta\ \rangle$ *where* $x$ *occurs in* $\vec{u}$.

**Interaction:**

$$\langle\ \vec{u}\ \mid\ \alpha(t_1, \ldots, t_n) = \beta(s_1, \ldots, s_m), \Theta\ \rangle$$

$$\rightarrow_{\mathrm{int}} \langle\ \vec{u}\ \mid\ \widehat{\Delta}[t_1/x_1, \ldots, t_n/x_n, s_1/y_1, \ldots, s_m/y_m], \Theta\ \rangle$$

*where* $\alpha(x_1, \ldots, x_n) = \beta(y_1, \ldots, y_m) \Rightarrow \Delta \in \mathcal{R}_{\mathrm{lt}}$.

We use just $\rightarrow$ instead of $\rightarrow_{\mathrm{com}}, \rightarrow_{\mathrm{sub}}, \rightarrow_{\mathrm{col}}, \rightarrow_{\mathrm{int}}$ when there is no ambiguity. We define $C_1 \Downarrow C_2$ by $C_1 \rightarrow^* C_2$ where $C_2$ is in normal form.

**Example 4.2.5**

*Rules in Figure 2.1 can be represented as follows:*

$$\begin{aligned} \mathtt{Add}(x_1, x_2) = \mathtt{S}(y) &\ \Rightarrow\ \mathtt{Add}(x_1, w) = y, x_2 = \mathtt{S}(w) \\ \mathtt{Add}(x_1, x_2) = \mathtt{Z} &\ \Rightarrow\ x_1 = x_2 \end{aligned}$$

*and the following computation can be performed:*

$$\begin{aligned} \langle\ r\ \mid\ \mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(\mathtt{Z})\ \rangle &\ \rightarrow_{\mathrm{int}}\ \langle\ r\ \mid\ \mathtt{Add}(\mathtt{Z}, w') = \mathtt{Z}, r = \mathtt{S}(w')\ \rangle \\ &\ \rightarrow_{\mathrm{int}}\ \langle\ r\ \mid\ \mathtt{Z} = w', r = \mathtt{S}(w')\ \rangle \\ &\ \rightarrow_{\mathrm{col}}\ \langle\ \mathtt{S}(w')\ \mid\ \mathtt{Z} = w'\ \rangle \\ &\ \rightarrow_{\mathrm{col}}\ \langle\ \mathtt{S}(\mathtt{Z})\ \mid\ \rangle. \end{aligned}$$

### 4.2.4 Properties of lightweight reduction rules

In this section we present some properties of the lightweight reduction rules. First, we show that we can postpone applications of Collect rule as in Abramsky's Computational interpretations of linear logic [1].

**Lemma 4.2.6**

*If* $C_1 \to_{\mathrm{col}} \cdot \to_{\mathrm{com}} C_2$, *then* $C_1 \to_{\mathrm{com}} \cdot \to_{\mathrm{col}} C_2$.

**Proof.** By applying the Collect rule, no equations are caused such that the Communication rule can be applied. Thus, the result can become the same even if the Communication rule is applied before the Collect rule. For instance, we assume that $C_1$ is $\langle\ \vec{u}\ \mid\ x = t, y = s, y = u, \Delta\ \rangle$ where $x$ occurs in $\vec{u}$, then the following holds since $x$ does not occur in $s, u$ and $\Delta$:

$$\begin{array}{ccc}
\langle\ \vec{u}\ \mid\ x = t, y = s, y = u, \Delta\ \rangle & \xrightarrow[\mathrm{com}]{} & \langle\ \vec{u}\ \mid\ x = t, s = u, \Delta\ \rangle \\
\Big\downarrow{\scriptstyle \mathrm{col}} & & \Big\downarrow{\scriptstyle \mathrm{col}} \\
\langle\ \vec{u}[t/x]\ \mid\ y = s, y = u, \Delta\ \rangle & \xrightarrow[\mathrm{com}]{} & \langle\ \vec{u}[t/x]\ \mid\ s = u, \Delta\ \rangle
\end{array}$$

□

**Lemma 4.2.7**

*If* $C_1 \to_{\mathrm{col}} \cdot \to_{\mathrm{sub}} C_2$, *then* $C_1 \to_{\mathrm{sub}} \cdot \to_{\mathrm{col}} C_2$.

**Proof.** Because there is no equation that overlaps with the Collect and Substitute rules, the result can become the same regardless of the order of applying these rules.

$$\begin{array}{ccc}
\langle\ \vec{u}\ \mid\ x = t, y = s, \Delta\ \rangle & \xrightarrow[\mathrm{sub}]{} & \langle\ \vec{u}\ \mid\ x = t, \Delta[s/y]\ \rangle \\
\Big\downarrow{\scriptstyle \mathrm{col}} & & \Big\downarrow{\scriptstyle \mathrm{col}} \\
\langle\ \vec{u}[t/x]\ \mid\ y = s, \Delta\ \rangle & \xrightarrow[\mathrm{sub}]{} & \langle\ \vec{u}[t/x]\ \mid\ \Delta[s/y]\ \rangle
\end{array}$$

□

**Lemma 4.2.8**

*If* $C_1 \to_{\mathrm{col}} \cdot \to_{\mathrm{int}} C_2$, *then* $C_1 \to_{\mathrm{int}} \cdot \to_{\mathrm{col}} C_2$.

**Proof.** Because there is no equation that is overlapped with Collect rule and Interaction rule, the result can become the same regardless order of applying these rules.

$$\begin{array}{ccc} \langle\ \vec{u} \mid x = t, s_1 = s_2, \Delta\ \rangle & \xrightarrow[\text{int}]{} & \langle\ \vec{u} \mid x = t, \Delta', \Delta\ \rangle \\ \Big\downarrow{\scriptstyle \text{col}} & & \Big\downarrow{\scriptstyle \text{col}} \\ \langle\ \vec{u}[t/x] \mid s_1 = s_2, \Delta\ \rangle & \xrightarrow[\text{int}]{} & \langle\ \vec{u}[t/x] \mid \Delta', \Delta\ \rangle \end{array}$$

□

By Lemmas 4.2.6, 4.2.7 and 4.2.8, the following holds.

**Theorem 4.2.9**

*Let $C_1 \Downarrow C_2$. Then there is a configuration $C$ such that $C_1 \to^* C \to^*_{\text{col}} C_2$ and $C_1$ is reduced to $C$ without application of any* Collect *rule.* □

Next, we examine whether we can postpone the application of Substitution rules or not. Note that applying the Substitution rule to an equation does not generate any other equations which require an application of the Interaction rule. Therefore the following properties hold.

**Lemma 4.2.10**

*If $C_1 \to_{\text{sub}} \cdot \to_{\text{com}} C_2$, then $C_1 \to_{\text{com}} \cdot \to_{\text{sub}} C_2$.*

**Proof.**

$$\begin{array}{ccc} \langle\ \vec{u} \mid x = s, \alpha(\vec{t}) = y, y = t, \Delta\ \rangle & \xrightarrow[\text{com}]{} & \langle\ \vec{u} \mid x = s, \alpha(\vec{t}) = t, \Delta\ \rangle \\ \Big\downarrow{\scriptstyle \text{sub}} & & \Big\downarrow{\scriptstyle \text{sub}} \\ \langle\ \vec{u} \mid \alpha(\vec{t})[s/x] = y, y = t, \Delta\ \rangle & \xrightarrow[\text{com}]{} & \langle\ \vec{u} \mid \alpha(\vec{t})[s/x] = t, \Delta\ \rangle \end{array}$$

□

**Lemma 4.2.11**

*If $C_1 \to_{\text{sub}} \cdot \to_{\text{int}} C_2$, then $C_1 \to_{\text{int}} \cdot \to_{\text{sub}} C_2$ or $C_1 \to_{\text{int}} \cdot \to_{\text{com}} C_2$.*

**Proof.** For an application of the Substitution rule in $C_1 \to_{\text{sub}} \cdot \to_{\text{int}} C_2$, we assume that, as the case that the equations are overlapped, there is an equation $x = t$ in $C_1$ such that the $x$ is occurred in a term $s$ which is managed by the Interaction rule.

After applying the Interaction rule in $C_1$, there are two possibilities. One is that there are equations which have the term $s$ as well. In this case, we can have the same result $C_2$ by the application of the Substitution rule. The other is that there is an equation whose LHS or RHS is just the $x$. By the Communication rule for $x$, we can have the same result as well. □

By Theorem 4.2.9, and Lemmas 4.2.10 and 4.2.11, the following theorem holds.

**Theorem 4.2.12**

*If $C_1 \Downarrow C_2$, then there is a configuration $C$ such that $C_1 \to^* C \to^*_{\mathrm{sub}} \to^*_{\mathrm{col}} C_2$ and $C_1$ is reduced to $C$ by applying only* Interaction *and* Communication *rules.* □

This theorem shows that all Interaction rules can be performed without application of Substitution rules. We define $C_1 \Downarrow_{\mathrm{ic}} C_2$ by $C_1 \to^* C_2$ where $C_2$ is a $\{\to_{\mathrm{int}}, \to_{\mathrm{com}}\}$-normal form. Because all critical pairs that are generated by $\to_{\mathrm{int}}$ and $\to_{\mathrm{com}}$ are confluent, the determinacy property holds:

**Theorem 4.2.13 (Determinacy)**

*Let $C_1 \Downarrow C_2$. When there are configurations $C', C''$ such that $C_1 \Downarrow_{\mathrm{ic}} C'$ and $C_1 \Downarrow_{\mathrm{ic}} C''$, then $C'$ is equivalent to $C''$.* □

## 4.3 Simpler lightweight abstract machine

In this section we adapt the lightweight abstract machine so that a connection between names can be represented by single link. This is realised by allowing the environment to contain a mapping of names to names which was not possible in the lightweight abstract machine.

We define a configuration of our abstract machine state by the following 4-tuple

$$(\,\mathrm{E} \mid \vec{t} \mid H \mid \Gamma\,)$$

where

- E is an environment, which is a set $\mathcal{N} \times \mathcal{T}$ (where $\mathcal{N}$ is a set of names and $\mathcal{T}$ is the set of terms),
- $\vec{t}$ is an interface, which is a sequence of terms,
- $H$ is a sequence of equations that are not executable,
- $\Gamma$ is a sequence of equations to operate.

In comparison to the SECD machine [37], the stack $S$, the environment $E$ and the control $C$ in the machine are corresponding to the term sequence $\vec{t}$, the map E, and the equation sequence $\Gamma$ in this abstract machine respectively. There is no element corresponding to the dump $D$ in the SECD machine because, during an execution of a rule, other rules are not called.

In Figure 4.1 we give the semantics of the machine as a set of transitional rules of the form: $(\,\mathrm{E} \mid \vec{t} \mid H \mid \Gamma\,) \Longrightarrow (\,\mathrm{E}' \mid \vec{t} \mid H' \mid \Gamma'\,)$.

| | | Before | After |
|---|---|---|---|
| A | Error | $H$ | $\mathsf{Error}(\alpha(\vec{t}) = \beta(\vec{s}))$, $H$ |
| | Code | $\alpha(\vec{t}) = \beta(\vec{s})$, $\Gamma$ | $\mathsf{Interaction}(\alpha(\vec{t}) = \beta(\vec{s}))$, $\Gamma$ |
| B.1 | Env. | $\mathrm{E}[x] = \bot$ | $\mathrm{E}[x] := t$ |
| | Code | $x = t, \Gamma$ | $\Gamma$ |
| B.2 | Env. | $\mathrm{E}[x] = \bot$ | $\mathrm{E}[x] := t$ |
| | Code | $t = x, \Gamma$ | $\Gamma$ |
| C.1 | Env. | $\mathrm{E}[x] = s$ | $\mathrm{E}[x] := \bot$ |
| | Code | $x = t, \Gamma$ | $s = t$, $\Gamma$ |
| C.2 | Env. | $\mathrm{E}[x] = s$ | $\mathrm{E}[x] := \bot$ |
| | Code | $t = x, \Gamma$ | $t = s$, $\Gamma$ |

Figure 4.1: Transitions $(\,\mathrm{E} \mid \vec{u} \mid H \mid \Gamma\,) \Longrightarrow (\,\mathrm{E}' \mid \vec{u} \mid H' \mid \Gamma'\,)$

To aid readability we present the transitions in a table format. For example, the entry:

| | | Before | After |
|---|---|---|---|
| C.1 | Env. | $\mathrm{E}[x] = s$ | $\mathrm{E}[x] := \bot$ |
| | Code | $x = t, \Gamma$ | $s = t$, $\Gamma$ |

corresponds to: $(\,\mathrm{E}[x] = s \mid \vec{u} \mid H \mid x = t,\, \Gamma\,) \Longrightarrow (\,\mathrm{E}[x] := \bot \mid \vec{u} \mid H \mid s = t,\, \Gamma\,)$.

Next, we define a compilation from a configuration to a machine state.

**Definition 4.3.1 (Compilation)**

*We define a translation* Compile *from a configuration into a machine state as follows:*

$$\mathsf{Compile}(\langle\ \vec{u}\ \mid\ \Delta\ \rangle) \stackrel{\text{def}}{=} (\,\emptyset \mid \vec{u} \mid - \mid \Gamma\,)$$

*where* $\Gamma$ *is a sequence of equations that is the result of fixing an order of the multiset of equations* $\Delta$*.*

The execution result is obtained by using the following Update operation:

**Definition 4.3.2**

*We define an operation* Update *for a machine state as follows:*

- $\mathsf{Update}(\,\{(x, s)\} \cup \mathrm{E} \mid \vec{u} \mid H \mid \Gamma\,)$

$$= \begin{cases} \mathsf{Update}((\,\mathrm{E}[s/x] \mid \vec{u}[s/x] \mid H[s/x] \mid \Gamma[s/x]\,)) & \textit{(when } x \textit{ occurs in } \mathrm{E}, \vec{u}, H \textit{ or } \Gamma\textit{)} \\ \mathsf{Update}(\,(\mathrm{E} \mid \vec{u} \mid x = s,\, H \mid \Gamma\,) & \textit{(otherwise)} \end{cases}$$

- $\mathsf{Update}(\,\emptyset \mid \vec{u} \mid H \mid \Gamma\,) = (\,\emptyset \mid \vec{u} \mid H \mid \Gamma\,)$.

**Example 4.3.3**

*The computation of the configuration* $\langle\ r\ \mid\ \mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(\mathtt{Z})\ \rangle$ *in Figure 2.1 is given below:*

$$
\begin{aligned}
&(\ \emptyset \mid r \mid - \mid \mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(\mathtt{Z})\ ) \\
&\quad \Longrightarrow (\ \emptyset \mid r \mid - \mid \mathtt{Add}(\mathtt{Z}, x) = \mathtt{Z}, r = \mathtt{S}(x)\ ) && \text{(A)} \\
&\quad \Longrightarrow (\ \emptyset \mid r \mid - \mid \mathtt{Z} = x, r = \mathtt{S}(x)\ ) && \text{(A)} \\
&\quad \Longrightarrow (\ \{(x, \mathtt{Z})\} \mid r \mid - \mid r = \mathtt{S}(x)\ ) && \text{(B.2)} \\
&\quad \Longrightarrow (\ \{(x, \mathtt{Z}), (r, \mathtt{S}(x))\} \mid r \mid - \mid -\ ). && \text{(B.1)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Update}(\ \{(x, \mathtt{Z}), (r, \mathtt{S}(x))\} \mid r \mid - \mid -\ ) \\
&\quad = \mathsf{Update}(\ \{(r, \mathtt{S}(\mathtt{Z}))\} \mid r \mid - \mid -\ ) \\
&\quad = \mathsf{Update}(\ \emptyset \mid \mathtt{S}(\mathtt{Z}) \mid - \mid -\ ) \\
&\quad = \langle\ \mathtt{S}(\mathtt{Z})\ \mid\ \rangle.
\end{aligned}
$$

### 4.3.1 Correctness

In order to show the correctness of our abstract machine for the lightweight calculus, we first define a decompilation function from machine states to configurations. Several lemmas follow before the correctness theorem.

**Definition 4.3.4 (Decompilation)**

- *We define a translation* $\mathsf{ToConfig}_e$ *from an environment* E *into a multiset of equations as follows:*

$$
\begin{aligned}
\mathsf{ToConfig}_e(\emptyset) &\stackrel{\text{def}}{=} -, \\
\mathsf{ToConfig}_e(\mathrm{E}[x] = t) &\stackrel{\text{def}}{=} x = t,\ \mathsf{ToConfig}_e(\mathrm{E}[x] := \bot).
\end{aligned}
$$

- *A translation* $\mathsf{ToConfig}_m$ *translates a machine state into a configuration as follows:*

$$
\mathsf{ToConfig}_m(\ \mathrm{E} \mid \vec{u} \mid H \mid \Gamma\ ) \stackrel{\text{def}}{=} \langle\ \vec{u}\ \mid\ \mathsf{ToConfig}_e(\mathrm{E}),\ H,\ \Gamma\ \rangle.
$$

- *We write just* ToConfig *instead of* $\mathsf{ToConfig}_e$, $\mathsf{ToConfig}_m$ *when there is no ambiguity.*

The machine will stop when there is no executable code. We define consistency of a machine state to ensure that each component has an appropriate piece of a configuration:

**Definition 4.3.5 (Consistency of a machine state)**

*A state* $(\ \mathrm{E} \mid \vec{u} \mid H \mid \Gamma\ )$ *is consistent iff*

- $\mathsf{ToConfig}(\ \mathrm{E} \mid \vec{u} \mid H \mid \Gamma\ )$ *is a configuration, thus every name occurs at most twice,*
- $\langle\ \mid\ H\ \rangle$ *is a normal form,*
- *for every* $x \in \mathcal{N}$*, there is no elements such that* $(x, t_1), (x, t_2)$ *in* E *where* $t_1 \neq t_2$.

The following lemma shows that consistency is preserved during transitions:

**Lemma 4.3.6**

*Let $M_1$ be a consistent state. If $M_1 \Longrightarrow M_2$, then $M_2$ is also consistent.* □

**Lemma 4.3.7**

*Let $M_1$ be a consistent state. If $M_1 \Longrightarrow M_2$, then one of the following holds:*

- $\mathsf{ToConfig}(M_1) = \mathsf{ToConfig}(M_2)$,
- $\mathsf{ToConfig}(M_1) \rightarrow_{\mathrm{int}} \mathsf{ToConfig}(M_2)$,
- $\mathsf{ToConfig}(M_1) \rightarrow_{\mathrm{com}} \mathsf{ToConfig}(M_2)$.

**Proof.** We check each transition rule.

A: $M_1 = (\,\mathrm{E} \mid \vec{u} \mid H \mid \alpha(\vec{t}) = \beta(\vec{s}), \Gamma\,)$

$\Longrightarrow (\,\mathrm{E} \mid \vec{u} \mid \mathsf{Error}(\alpha(\vec{t}) = \beta(\vec{s})), H \mid \mathsf{Interaction}(\alpha(\vec{t}) = \beta(\vec{s})), \Gamma\,) = M_2$. If there is an interaction rule for the pair $(\alpha, \beta)$, then $\mathsf{ToConfig}(M_1) \rightarrow_{\mathrm{int}} \mathsf{ToConfig}(M_2)$. Otherwise, $\mathsf{ToConfig}(M_1) = \mathsf{ToConfig}(M_2)$.

B.1: $M_1 = (\,\mathrm{E} \mid \vec{u} \mid H \mid x = t, \Gamma\,) \Longrightarrow (\,\mathrm{E}[x] := t \mid \vec{u} \mid H \mid \Gamma\,) = M_2$. By Definition 4.3.4, $\mathsf{ToConfig}(M_1) = \mathsf{ToConfig}(M_2)$.

B.2: The same as the case of B.1.

C.1: $M_1 = (\,\mathrm{E}[x] = s \mid \vec{u} \mid H \mid x = t, \Gamma\,) \Longrightarrow (\,\mathrm{E}[x] := \bot \mid \vec{u} \mid H \mid s = t, \Gamma\,) = M_2$.

$\mathsf{ToConfig}(M_1) = \langle\ \vec{u} \mid x = s, \mathsf{ToConfig}(\mathrm{E}), H, x = t, \Gamma\ \rangle$

$\rightarrow_{\mathrm{com}} \langle\ \vec{u} \mid \mathsf{ToConfig}(\mathrm{E}), H, s = t, \Gamma\ \rangle = \mathsf{ToConfig}(M_2)$.

C.2: The same as the case of C.1. □

**Lemma 4.3.8**

*Let $M_1$ be a consistent state. If $M_1 \Downarrow (\,\mathrm{E} \mid \vec{u} \mid H \mid \Gamma\,)$, then $\Gamma$ is empty.*

**Proof.** There exists a transition which can be applied to an equation in $\Gamma$ whenever machine states are consistent. □

Let $M_1$ and $M_2$ be two machine states. We define $M_1 \Downarrow M_2$ by $M_1 \Longrightarrow^* M_2$ where $M_2$ is a $\Longrightarrow$-normal form.

With respect to $\{\rightarrow_{\mathrm{int}}, \rightarrow_{\mathrm{com}}\}$-normal forms, by Lemmas 4.3.7, 4.3.8, the following correctness holds:

**Theorem 4.3.9 (Correctness in $\{\rightarrow_{\mathrm{int}}, \rightarrow_{\mathrm{com}}\}$-normal forms)**
*Let $C$ be a configuration. If* $\mathsf{Compile}(C) \Downarrow (\mathrm{E} \mid \vec{u} \mid H \mid \Gamma)$, *then $\Gamma$ is empty and* $C \Downarrow_{ic} \mathsf{ToConfig}(\mathrm{E} \mid \vec{u} \mid H \mid -)$. □

In addition, the completeness also holds:

**Theorem 4.3.10 (Completeness in $\{\rightarrow_{\mathrm{int}}, \rightarrow_{\mathrm{com}}\}$-normal forms)**
*Let $C_1, C_2$ be configurations such that $C_1 \Downarrow_{ic} C_2$. Then there is a machine state $M$ such that* $\mathsf{Compile}(C_1) \Downarrow M$ *and* $\mathsf{ToConfig}(M) = C_2$.

**Proof.** If $\mathsf{Compile}(C_1)$ has no normal forms, corresponding to an infinite transition sequence starting from $\mathsf{Compile}(C_1)$ we can construct an infinite reduction sequence starting from $C_1$ by Lemma 4.3.7 since the numbers of equations produced in each transition are finite. This contradicts the assumption of this theorem, and thus there is an $M$ such that $\mathsf{Compile}(C_1) \Downarrow M$. By Theorem 4.3.9, $C_1 \Downarrow_{ic} \mathsf{ToConfig}(M)$, and therefore $\mathsf{ToConfig}(M) = C_2$ by the determinacy (Theorem 4.2.13). □

By Lemmas 4.3.12, 4.3.13 proved later, the following correctness property holds:

**Theorem 4.3.11 (Correctness)**
*Let $C$ be a configuration. If* $\mathsf{Compile}(C) \Downarrow M$, *then* $C \Downarrow \mathsf{ToConfig}(\mathsf{Update}(M))$. □

**Lemma 4.3.12**
*Let $\Gamma_n$ be a sequence $x_1 = t_1, x_2 = t_2, \ldots, x_n = t_n$ where $x_i \neq x_j (1 \leq i, j \leq n,\ i \neq j)$, $\vec{u}$ be an interface and $C_n = \langle\ \vec{u} \mid \Gamma_n\ \rangle$. Then there is a configuration $C'$ such that $C_n \Downarrow C'$ by applying rules except for the Interaction rule.*

**Proof.** By induction on the number $n$ of the equations.

- In the case of $n = 1$: Let $\Gamma_1$ be $x_1 = t_1$.
  - When $x_1$ does not occur twice in $C_1$: There is no rule that can apply to $C_1$
  - Otherwise: $x_1$ occurs in either $t_1$ or $\vec{u}$.
    - In the case of $t_1$, there is no rule that can apply to $C_1$.
    - In the case of $\vec{u}$, $C_1 = \langle\ \vec{u} \mid x_1 = t_1\ \rangle \rightarrow_{\mathrm{col}} \langle\ \vec{u}[t_1/x_1] \mid\ \rangle$.
- In the case of $n = 2$: Let $\Gamma_2$ be $x_1 = t_1, x_2 = t_2$ where $x_1 \neq x_2$.
  - When $x_2$ does not occur twice in $C_2$: There is no rule that can apply to $C_2$.
  - Otherwise: $x_2$ occurs in one of $t_2, t_1$ and $\vec{u}$.
    - In the case of $t_2$: There is no rule that can apply to $C_2$.

  - In the case of $t_1$:

    $C_2 = \langle\ \vec{u}\ \mid\ x_1 = t_1, x_2 = t_2\ \rangle \to \langle\ \vec{u}\ \mid\ x_1 = t_1[t_2/x_2]\ \rangle = C_1$ by $\to_{\mathrm{sub}}$ or $\to_{\mathrm{com}}$.

  - In the case of $\vec{u}$:

    $C_2 = \langle\ \vec{u}\ \mid\ x_1 = t_1, x_2 = t_2\ \rangle \to_{\mathrm{col}} \langle\ \vec{u}[t_2/x_2]\ \mid\ x_1 = t_1\ \rangle$. By applying the result of the case of $n = 1$, it holds.

- In the case of $n = k + 1$:

  In the similar way of the case of $n = 2$, the number of equations can be decreased. Thus, by applying the induction hypothesis, it holds. □

The following shows that each execution of Update corresponds to an application of either Substitution or Collect rules:

**Lemma 4.3.13**

*Let $M$ be a consistent normal form. Then* $\mathsf{ToConfig}(M) \Downarrow \mathsf{ToConfig}(\mathsf{Update}(M))$ *by applying rules except for the Interaction rule.*

**Proof.** We assume that $M$ is $(\,\mathrm{E}\mid\vec{u}\mid H\mid\Gamma\,)$. Since $\Gamma$ is empty by Lemma 4.3.8, $M = (\,\mathrm{E}\mid\vec{u}\mid H\mid -\,)$.

By induction on the number $n$ of elements in E:

- In the case of $n = 0$: Trivial.

- In the case of $n = 1$:

  Let $\mathrm{E} = \{(x, s)\}$. Then $M = (\,\{(x, s)\}\mid\vec{u}\mid H\mid -\,)$ and $\mathsf{ToConfig}(M) = \langle\ \vec{u}\ \mid\ x = s, H\ \rangle$. Because $M$ is consistent, $x$ occurs once in one of $\vec{u}, H$ and $s$, or not at all.

  - In the case of $\vec{u}$:

    $\mathsf{Update}(M) = (\,\emptyset\mid\vec{u}[s/x]\mid H\mid -\,)$ and $\mathsf{ToConfig}(\mathsf{Update}(M)) = \langle\ \vec{u}[s/x]\ \mid\ H\ \rangle$. Thus, $\mathsf{ToConfig}(M) \to_{\mathrm{col}} \mathsf{ToConfig}(\mathsf{Update}(M))$. Because there is no rule that can apply to $H$, $\mathsf{ToConfig}(\mathsf{Update}(M))$ is a $\to$-normal form, and therefore $\mathsf{ToConfig}(M) \Downarrow \mathsf{ToConfig}(\mathsf{Update}(M))$.

  - In the case of $H$:

    $\mathsf{Update}(M) = (\,\emptyset\mid\vec{u}\mid H[s/x]\mid -\,)$ and $\mathsf{ToConfig}(\mathsf{Update}(M)) = \langle\ \vec{u}\ \mid\ H[s/x]\ \rangle$. Thus, $\mathsf{ToConfig}(M) \to_{\mathrm{sub}} \mathsf{ToConfig}(\mathsf{Update}(M))$. Because there is no rule that can apply to $H[s/x]$, $\mathsf{ToConfig}(\mathsf{Update}(M))$ is a $\to$-normal form.

  - In the case of $s$:

    $\mathsf{Update}(M) = (\,\emptyset \mid \vec{u} \mid x = s, H \mid -\,)$ and $\mathsf{ToConfig}(\mathsf{Update}(M)) = \langle\ \vec{u} \mid x = s,\ H\ \rangle$. Thus, $\mathsf{ToConfig}(M) = \mathsf{Update}(M)$. Because there is no rule that can apply to $x = s$, $\mathsf{ToConfig}(\mathsf{Update}(M))$ is a $\rightarrow$-normal form.

  - In the case that $x$ does not occur in $\vec{u}, H$ and $s$: $\mathsf{Update}(M) = (\,\emptyset \mid \vec{u} \mid x = s, H \mid -\,)$ and $\mathsf{ToConfig}(\mathsf{Update}(M)) = \langle\ \vec{u} \mid x = s,\ H\ \rangle$. This is proven in the similar way to the case of $s$.

- In the case of $n = k + 1$:

  Let $\mathrm{E}_{k+1} = \{(x, s)\} \cup \mathrm{E}_k$. Then $\mathsf{ToConfig}(M) = \langle\ \vec{u} \mid x = s, \mathsf{ToConfig}(\mathrm{E}_k), H\ \rangle$.

  Because $M$ is consistent, $x$ occurs once in one of $\vec{u}, H, s$ and $\mathsf{ToConfig}(\mathrm{E}_k)$, or not at all:

  - In the case of one of $\vec{u}, H$ and $s$, or not at all:

    In the similar way of the case of $n = 1$, the number of equations can be decreased by applying rules except for the Interaction rule. Thus, by applying the induction hypothesis, it holds.

  - In the case of $\mathsf{ToConfig}(\mathrm{E}_k)$:

    By Lemma 4.3.12, the number of equations can be decreased by applying rules except the Interaction rule. Thus, by applying the induction hypothesis, it holds. □

### 4.3.2 Computation without the map for connections

In this section we show an example such that the computation using the map for connections in the lightweight abstract machine can be performed in the simpler lightweight abstract machine without the map.

We take the following sequence of equations that requires the map for connections to be performed in the lightweight abstract machine:

$$x = y,\ y = \alpha,\ x = \beta.$$

The lightweight abstract machine reduces it to $\alpha = \beta$ by three steps:

$$\begin{aligned}
&(\,\mathrm{E} \mid \mathrm{P} \mid - \mid - \mid x = y,\ y = \alpha,\ x = \beta\,) \\
&\Longrightarrow (\,\mathrm{E} \mid \mathrm{P}[x \leftrightarrow y] \mid - \mid - \mid y = \alpha,\ x = \beta\,) && \text{(III.0_0)} \\
&\Longrightarrow (\,\mathrm{E}[x \mapsto \alpha] \mid \mathrm{P} \mid - \mid - \mid x = \beta\,) && \text{(II.e)}
\end{aligned}$$

$$\Longrightarrow (\,\mathrm{E} \mid \mathrm{P} \mid - \mid - \mid \alpha = \beta\,). \tag{II.c}$$

The simpler lightweight abstract machine allows the environment to contain a mapping of names to names which was not possible in the lightweight abstract machine, and it reduces the sequence without the map as follows:

$$(\,\mathrm{E} \mid - \mid - \mid x = y,\ y = \alpha,\ x = \beta\,)$$
$$\Longrightarrow (\,\mathrm{E}[x \mapsto y] \mid - \mid - \mid y = \alpha,\ x = \beta\,) \tag{B.1}$$
$$\Longrightarrow (\,\mathrm{E}[x \mapsto y][y \mapsto \alpha] \mid - \mid - \mid x = \beta\,) \tag{B.1}$$
$$\Longrightarrow (\,\mathrm{E}[y \mapsto \alpha] \mid - \mid - \mid y = \beta\,) \tag{C.1}$$
$$\Longrightarrow (\,\mathrm{E} \mid - \mid - \mid \alpha = \beta\,). \tag{C.1}$$

Although the simpler lightweight abstract machine is correct for the lightweight calculus, the reduction sequence tends to be longer than the sequence in the lightweight abstract machine. This is caused by the absence of a map for names, and thus in equations such as $x = y$ only the LHS names $x$ are managed. In the above example, the computation of the equation $x = y$ is managed only the LHS name $x$, and thus $y = \alpha$ is accumulated in the environment on the third line, while in the lightweight abstract machine the $y$ is managed using information of the map $\mathrm{P}[x \leftrightarrow y]$.

The method of computation without the map in the simpler lightweight abstract machine also requires the longer reduction sequence, when equations $x = y$ are managed, even if the map is not used in the lightweight abstract machine. For instance, we take the following sequence of equations:

$$x = \alpha,\ x = y,\ y = \beta.$$

While the lightweight abstract machine reduces it to $\alpha = \beta$ by three steps without using the map:

$$(\,\mathrm{E} \mid \mathrm{P} \mid - \mid - \mid x = \alpha,\ x = y,\ y = \beta\,)$$
$$\Longrightarrow (\,\mathrm{E}[x \mapsto \alpha] \mid \mathrm{P} \mid - \mid - \mid x = y,\ y = \beta\,) \tag{II.0}$$
$$\Longrightarrow (\,\mathrm{E}[y \mapsto \alpha] \mid \mathrm{P} \mid - \mid - \mid y = \beta\,) \tag{III.e\_0}$$
$$\Longrightarrow (\,\mathrm{E} \mid \mathrm{P} \mid - \mid - \mid \alpha = \beta\,), \tag{III.e}$$

the simpler lightweight abstract machine takes four steps:

$$(\,\mathrm{E} \mid - \mid - \mid x = \alpha,\ x = y,\ y = \beta\,)$$
$$\Longrightarrow (\,\mathrm{E}[x \mapsto \alpha] \mid - \mid - \mid x = y,\ y = \beta\,) \tag{B.1}$$
$$\Longrightarrow (\,\mathrm{E} \mid - \mid - \mid \alpha = y,\ y = \beta\,) \tag{C.1}$$
$$\Longrightarrow (\,\mathrm{E}[y \mapsto \alpha] \mid - \mid - \mid y = \beta\,) \tag{B.2}$$
$$\Longrightarrow (\,\mathrm{E} \mid - \mid - \mid \alpha = \beta\,). \tag{C.1}$$

This is because in the single link encoding method only a single side of an equation is managed, while in the amineLight encoding both sides of an equation such as $x = y$ are managed using the map. The effect to the execution performance will be discussed in Section 4.5.3.

## 4.4 Simpler textual calculus

In this section we introduce a simpler textual calculus as a fusion with the simpler abstract machine by introducing a term $\$t$, which is called an *indirection term*, to show that the Environment E in a machine state has a pair $(x, t)$, and by substituting $\$t$ for the name $x$. This notation is closer to the implementation method.

First, we introduce the indirection term $\$t$ into the grammar definition of terms.

**Definition 4.4.1 (Indirection terms)**

- *We extend terms on $\Sigma$ and $\mathcal{N}$ by the following grammar:*

$$t ::= x \mid \alpha(t_1, \ldots, t_n) \mid \$t$$

  *where $x \in \mathcal{N}$, $\alpha \in \Sigma$, $\mathsf{ar}(\alpha) = n$ and $t_1, \ldots, t_n$ are terms. We call the terms $\$t$* indirection terms. *When terms do not contain indirection terms, we call the terms* indirection free.

- *We define a configuration as a pair: $(\vec{t} \mid \Gamma)$, where $\vec{t}$ is a sequence $t_1, \ldots, t_n$ of terms, and $\Gamma$ is a sequence of equations. Each variable occurs at most twice in a configuration. Configurations that differ only on names are considered equivalent.*

According to this extension, the set of names in a term is also extended:

**Definition 4.4.2 (Names in terms)**

*The set $\mathsf{Name}(t)$ of names of a term $t$ is defined in the following way, which extends to sequences of terms, equations, sequences of equations, and rules in the obvious way.*

$$\begin{aligned} \mathsf{Name}(x) &= \{x\}, \\ \mathsf{Name}(\alpha(t_1, \ldots, t_n)) &= \mathsf{Name}(t_1) \cup \cdots \cup \mathsf{Name}(t_n), \\ \mathsf{Name}(\$t) &= \mathsf{Name}(t). \end{aligned}$$

**Definition 4.4.3 (Computation Rules)**

*The operational behaviour of the system is given by the following set of computation rules.*

**Interaction:** $(\vec{u} \mid \alpha(t_1, \ldots, t_n) = \beta(s_1, \ldots, s_m), \Gamma)$

$$\rightsquigarrow (\vec{u} \mid \widehat{\Gamma_1}[t_1/x_1, \ldots, t_n/x_n, s_1/y_1, \ldots, s_m/y_m], \Gamma)$$

$$\textit{where } \alpha(x_1, \ldots, x_n) = \beta(y_1, \ldots, y_m) \Rightarrow \Gamma_1 \in \mathcal{R}_{\mathrm{lt}}.$$

**Var1:** $(\,\vec{u} \mid x = t, \Gamma\,) \rightsquigarrow (\,\vec{u} \mid \Gamma\,)[\$t/x]$ *where* $t \neq \$s$.

**Var2:** $(\,\vec{u} \mid t = x, \Gamma\,) \rightsquigarrow (\,\vec{u} \mid \Gamma\,)[\$t/x]$ *where* $t \neq \$s$.

**Indirection1:** $(\,\vec{u} \mid \$t = s, \Gamma\,) \rightsquigarrow (\,\vec{u} \mid t = s, \Gamma\,)$.

**Indirection2:** $(\,\vec{u} \mid t = \$s, \Gamma\,) \rightsquigarrow (\,\vec{u} \mid t = s, \Gamma\,)$.

These rules will correspond directly to the graphical data structure and operations given in the next section. Indirection is introduced so that the data-structure manipulations can be kept simple. However, there is then the overhead of dealing with indirection nodes. We remark also that normal forms can have indirections in them, but only inside constructors. Computationally the interaction rule is the most expensive to implement: the other rules will turn out to be implemented with a small number of instructions or will be equivalences in the data structure. In particular, the renaming part of the hat operation is zero cost, as no renaming is needed in an implementation.

### 4.4.1 Expressive power

In this section we examine the expressive power of the simpler textual calculus.

Intuitively, a configuration $(\,\vec{u} \mid \Gamma\,)$ in the simpler textual calculus is regarded as an abbreviation of a machine state $(\,\mathrm{E} \mid \vec{u} \mid H \mid \Gamma\,)$ by ignoring the error code sequence $H$ and substituting $\$t$ for $(x, t) \in \mathrm{E}$. Moreover, each rewriting step in the calculus corresponds to a transition of a corresponding machine state since the transition rules Var1, Var2, Indirection1 and Indirection2 correspond to the machine transition rules B.1, B.2, C.1 and C.2 respectively.

In the following discussion, we assume that, for every active pair $\alpha(\vec{t}) = \beta(\vec{s})$ there is an interaction rule.

First, we define a translation from a machine state to a configuration in the simpler textual calculus.

**Definition 4.4.4**

*We define a translation* ToSimple *from a machine state to the simpler textual calculus as follows:*

$$\begin{aligned}
\mathsf{ToSimple}(\,\mathrm{E}[x] = t \mid \vec{u} \mid H \mid \Gamma\,) &\stackrel{\text{def}}{=} \mathsf{ToSimple}(\,\mathrm{E}[\$t/x] \mid \vec{u}[\$t/x] \mid H[\$t/x] \mid \Gamma[\$t/x]\,),\\
\mathsf{ToSimple}(\,\emptyset \mid \vec{u} \mid H \mid \Gamma\,) &\stackrel{\text{def}}{=} (\,\vec{u} \mid \Gamma\,).
\end{aligned}$$

We examine whether all transitions of machine states are performed in the simpler textual calculus.

**Lemma 4.4.5**

*Let $M_1$ and $M_2$ be consistent machine states such that $M_1 \Longrightarrow M_2$, then* $\mathsf{ToSimple}(M_1) \rightsquigarrow \mathsf{ToSimple}(M_2)$.

**Proof.** Let $M_1$ be $(\,\mathrm{E} \mid \vec{u} \mid H \mid \Gamma\,)$. We check each transition rule:

A: $M_1 = (\,\mathrm{E} \mid \vec{u} \mid H \mid t = s, \Gamma\,) \Longrightarrow (\,\mathrm{E} \mid \vec{u} \mid H \mid \Gamma_1, \Gamma\,) = M_2$.

Since $M_1$ and $M_2$ have the same E, the substitution performed by $\mathsf{ToSimple}$ is the same, $\mathsf{ToSimple}(M_1)$ and $\mathsf{ToSimple}(M_2)$ are written as follows:

$$\mathsf{ToSimple}(M_1) = (\,\vec{u} \mid t = s, \Gamma\,)[\$t_1/x_1, \ldots, \$t_n/x_n],$$
$$\mathsf{ToSimple}(M_2) = (\,\vec{u} \mid \Gamma_1, \Gamma\,)[\$t_1/x_1, \ldots, \$t_n/x_n].$$

Thus, $\mathsf{ToSimple}(M_1) \rightsquigarrow \mathsf{ToSimple}(M_2)$ by Interaction rule.

B.1: $M_1 = (\,\mathrm{E}[x] = \bot \mid \vec{u} \mid H \mid x = t, \Gamma\,) \Longrightarrow (\,\mathrm{E}[x] := t \mid \vec{u} \mid H \mid \Gamma\,) = M_2$.

When $\mathsf{ToSimple}(M_1) = (\,\vec{u'} \mid x = t', \Gamma'\,)$, then $\mathsf{ToSimple}(M_2) = (\,\vec{u'} \mid \Gamma'\,)[\$t'/x]$.

Thus, $\mathsf{ToSimple}(M_1) \rightsquigarrow \mathsf{ToSimple}(M_2)$ by Var1.

C.1: $M_1 = (\,\mathrm{E}[x] = s \mid \vec{u} \mid H \mid x = t, \Gamma\,) \Longrightarrow (\,\mathrm{E}[x] := \bot \mid \vec{u} \mid H \mid s = t, \Gamma\,) = M_2$

When $\mathsf{ToSimple}(M_1) = (\,\vec{u'} \mid x = t', \Gamma'\,)[\$s'/x]$, then $\mathsf{ToSimple}(M_2) = (\,\vec{u'} \mid s' = t', \Gamma'\,)$.

Thus, $\mathsf{ToSimple}(M_1) \rightsquigarrow \mathsf{ToSimple}(M_2)$ by Indirection1.

B.2 and C.2: Similar to B.1 and C.1 respectively. □

**Lemma 4.4.6**

*Let $M$ be a $\Longrightarrow$-normal form, then* $\mathsf{ToSimple}(M)$ *is also a normal form.*

**Proof.** We assume that $M$ is $(\,\mathrm{E} \mid \vec{u} \mid H \mid \Gamma\,)$. Since $\Gamma$ is empty by Lemma 4.3.8,

$$\mathsf{ToSimple}(\,\mathrm{E} \mid \vec{u} \mid H \mid -\,) = (\,\vec{u_1} \mid -\,)$$

for some term sequence $\vec{u_1}$, and thus $\mathsf{ToSimple}(M)$ is a normal form. □

Let $C_1$ and $C_2$ be configurations in the simpler textual calculus. We define $C_1 \Downarrow C_2$ by $C_1 \rightsquigarrow^* C_2$ where $C_2$ is a $\rightsquigarrow$-normal form.

By Lemmas 4.4.5 and 4.4.6, the following holds:

**Theorem 4.4.7 (Correctness for the simpler abstract machine)**

*Let $M_1$ and $M_2$ be consistent machine states such that $M_1 \Downarrow M_2$. Then* $\mathsf{ToSimple}(M_1) \Downarrow \mathsf{ToSimple}(M_2)$. □

By Theorems 4.3.9, 4.3.10 and 4.4.7, this calculus has enough rules to compute all of the interaction and communication rules.

Next, we examine whether all computation steps in the simpler textual calculus are performed in the simpler abstract machine.

**Definition 4.4.8**

*We define a translation* $\mathsf{ind2env}$ *from a term $t$ into a pair of an indirection free term $t'$ and an environment by induction on the structure of terms:*

$$\frac{}{\mathsf{ind2env}(x) = (x, \emptyset)}\ \mathit{name}$$

$$\frac{\mathsf{ind2env}(t_1) = (t'_1, \mathrm{E}_1) \quad \cdots \quad \mathsf{ind2env}(t_n) = (t'_n, \mathrm{E}_n)}{\mathsf{ind2env}(\alpha(t_1, \ldots, t_n)) = (\alpha(t'_1, \ldots, t'_n), \mathrm{E}_1 \cup \cdots \cup \mathrm{E}_n)}\ \mathit{agent}$$

$$\frac{\mathsf{ind2env}(t) = (t', \mathrm{E}_1)}{\mathsf{ind2env}(\$t) = (x, \{(x, t')\} \cup \mathrm{E}_1)}\ \mathit{indirection} \qquad \text{where } x \text{ is fresh.}$$

**Example 4.4.9**

*A term* $\$\mathtt{S}(\$\mathtt{Z})$ *is translated into* $(y, \{(y, \mathtt{S}(x)), (x, \mathtt{Z})\})$ *by* $\mathsf{ind2env}$ *as follows:*

$$\dfrac{\dfrac{\dfrac{\mathsf{ind2env}(\mathtt{Z}) = (\mathtt{Z}, \emptyset)}{\mathsf{ind2env}(\$\mathtt{Z}) = (x, \{(x, \mathtt{Z})\})}\ \mathit{indirection}}{\mathsf{ind2env}(\mathtt{S}(\$\mathtt{Z})) = (\mathtt{S}(x), \{(x, \mathtt{Z})\})}\ \mathit{agent}}{\mathsf{ind2env}(\$\mathtt{S}(\$\mathtt{Z})) = (y, \{(y, \mathtt{S}(x)), (x, \mathtt{Z})\})}\ \mathit{indirection}$$

**Definition 4.4.10**

*Let $(t_1, \mathrm{E}_1)$ and $(t_2, \mathrm{E}_2)$ be pairs of a term and an environment. The pairs $(t_1, \mathrm{E}_1)$ and $(t_2, \mathrm{E}_2)$ are called* $\alpha$-equivalent *when the following holds:*

$$(t_1, \mathrm{E}_1)[z_1/x_1, \ldots, z_n/x_n] = (t_2, \mathrm{E}_2)[z_1/y_1, \ldots, z_n/y_n]$$

*where $x_1, \ldots, x_n$ are names that occur in $(t_1, \mathrm{E}_1)$, $y_1, \ldots, y_n$ are names that occur in $(t_2, \mathrm{E}_2)$, and $z_1, \ldots, z_n$ are fresh names.*

**Lemma 4.4.11**

*Let $t$ be a term. When $(t_1, \mathrm{E}_1)$ and $(t_2, \mathrm{E}_2)$ are results of* $\mathsf{ind2env}(t)$*, then $(t_1, \mathrm{E}_1)$ and $(t_2, \mathrm{E}_2)$ are $\alpha$-equivalent.*

**Proof.** By induction on the structure of terms:

- When $t$ is $x$: $\mathsf{ind2env}(t) = (x, \emptyset)$, thus the result is defined uniquely.
- When $t$ is $\alpha(t_1, \ldots, t_n)$: We assume that, as the inductive hypothesis, for $1 \leq i \leq n$ $(t'_i, \mathrm{E}'_i)$ and $(t''_i, \mathrm{E}''_i)$ are results of $\mathsf{ind2env}(t_i)$ and $\alpha$-equivalent.

  Then, results of $\mathsf{ind2env}(\alpha(t_1, \ldots, t_n))$ are written as:

  $$(\alpha(t'_1, \ldots, t'_n), \mathrm{E}'_1 \cup \cdots \cup \mathrm{E}'_n) \quad (\alpha(t''_1, \ldots, t''_n), \mathrm{E}''_1 \cup \cdots \cup \mathrm{E}''_n).$$

  By the induction hypothesis, the results of $\mathsf{ind2env}(t)$ are $\alpha$-equivalent.
- When $t$ is $\$t_1$: We assume that, as the inductive hypothesis, $(t'_1, \mathrm{E}'_1)$ and $(t''_1, \mathrm{E}''_1)$ are results of $\mathsf{ind2env}(t_1)$ and $\alpha$-equivalent.

  Then, results of $\mathsf{ind2env}(\$t_1)$ are written as follows:

  $$(x, \{(x, t'_1)\} \cup \mathrm{E}'_1) \quad (y, \{(y, t''_1)\} \cup \mathrm{E}''_1)$$

  where $x$ and $y$ are fresh. These are written by using a fresh name $z$ as follows:

  $$(x, \{(x, t'_1)\} \cup \mathrm{E}'_1)[z/x] = (y, \{(y, t''_1)\} \cup \mathrm{E}''_1)[z/y]$$

  and by the induction hypothesis the results of $\mathsf{ind2env}(t)$ are $\alpha$-equivalent. □

When $(t_1, \mathrm{E}_1)$ and $(t_2, \mathrm{E}_2)$ are $\alpha$-equivalent, we may identify $(t_1, \mathrm{E}_1)$ and $(t_2, \mathrm{E}_2)$ as equivalent.

Next, we extend the definition of $\mathsf{ind2env}$ into sequences of terms, equations, sequences of equations and configurations.

**Definition 4.4.12**

*We extend the definition of* $\mathsf{ind2env}$ *into sequences of terms, equations, sequences of equations and configurations as follows:*

$$\frac{}{\mathsf{ind2env}(-) = (-, \emptyset)}\ \textit{empty}$$

$$\frac{\mathsf{ind2env}(t_1) = (t'_1, \mathrm{E}_1) \quad \cdots \quad \mathsf{ind2env}(t_n) = (t'_n, \mathrm{E_n})}{\mathsf{ind2env}(t_1, \ldots, t_n) = (t'_1, \ldots, t'_n, \mathrm{E}_1 \cup \cdots \cup \mathrm{E_n})}\ \textit{terms}$$

$$\frac{\mathsf{ind2env}(t) = (t', \mathrm{E}_1) \quad \mathsf{ind2env}(s) = (s', \mathrm{E}_2)}{\mathsf{ind2env}(t = s) = (t' = s', \mathrm{E}_1 \cup \mathrm{E}_2)}\ \textit{equation}$$

$$\frac{\mathsf{ind2env}(t_1 = s_1) = (t'_1 = s'_1, \mathrm{E}_1) \quad \cdots \quad \mathsf{ind2env}(t_n = s_n) = (t'_1 = s'_1, \mathrm{E}_n)}{\mathsf{ind2env}(t_1 = s_1, \ldots, t_n = s_n) = (t'_1 = s'_1, \ldots, t'_n = s'_n, \mathrm{E}_1 \cup \cdots \cup \mathrm{E}_n)}\ \textit{equations}$$

$$\frac{\mathsf{ind2env}(\vec{u}) = (\vec{u'}, \mathrm{E}_1) \quad \mathsf{ind2env}(\Gamma) = (\Gamma', \mathrm{E}_2)}{\mathsf{ind2env}(\ \vec{u} \mid \Gamma\ ) = (\ (\ \vec{u'} \mid \Gamma'\ ), \mathrm{E}_1 \cup \mathrm{E}_2)}\ \textit{configuration}$$

We extend the $\alpha$-equivalence on pairs of a term and an environment into pairs of a configuration and an environment:

**Definition 4.4.13**

*Let* $(C_1, \mathrm{E}_1)$ *and* $(C_2, \mathrm{E}_2)$ *be pairs of a configuration and an environment. The pairs* $(C_1, \mathrm{E}_1)$ *and* $(C_2, \mathrm{E}_2)$ *are called* $\alpha$-equivalent *when the following holds:*

$$(C_1, \mathrm{E}_1)[z_1/x_1, \ldots, z_n/x_n] = (C_2, \mathrm{E}_2)[z_1/y_1, \ldots, z_n/y_n]$$

*where* $x_1, \ldots, x_n$ *are names that occur in* $(C_1, \mathrm{E}_1)$, $y_1, \ldots, y_n$ *are names that occur in* $(C_2, \mathrm{E}_2)$, *and* $z_1, \ldots, z_n$ *are fresh names.*

**Lemma 4.4.14**

*Let* $C$ *be a configuration in the simpler textual calculus. When* $(C_1, \mathrm{E}_1)$ *and* $(C_2, \mathrm{E}_2)$ *are results of* $\mathsf{ind2env}(C)$, *then* $(C_1, \mathrm{E}_1)$ *and* $(C_2, \mathrm{E}_2)$ *are* $\alpha$*-equivalent.* $\square$

When $(C_1, \mathrm{E}_1)$ and $(C_2, \mathrm{E}_2)$ are $\alpha$-equivalent, we may identify $(C_1, \mathrm{E}_1)$ and $(C_2, \mathrm{E}_2)$ as equivalent.

Next, we define a translation from a configuration in the calculus to a machine state.

**Definition 4.4.15**

*Let* $C$ *be a configuration in the simpler textual calculus and* $\mathsf{ind2env}(C) = ((\,\vec{u} \mid \Gamma\,), \mathrm{E})$. *Then we define a translation* $\mathsf{ToState}$ *from the configuration to a machine state as follows:*

$$\mathsf{ToState}(C) \stackrel{\text{def}}{=} (\,\mathrm{E} \mid \vec{u} \mid - \mid \Gamma\,).$$

**Lemma 4.4.16**

*Let* $C_1$ *and* $C_2$ *be configurations in the simpler textual calculus such that* $C_1 \rightsquigarrow C_2$, *then* $\mathsf{ToState}(C_1) \Longrightarrow \mathsf{ToState}(C_2)$.

**Proof.** We check each computational rule in the calculus.

Interaction: We use a notation $\Gamma(x_1, \ldots, x_n)$ to show occurrences of free names $x_1, \ldots, x_n$ in a sequence of equations $\Gamma$ explicitly and a notation $\Gamma(t_1, \ldots, t_n)$ as the result obtained by substitution of terms $t_1, \ldots, t_n$ for the $x_1, \ldots, x_n$ in $\Gamma(x_1, \ldots, x_n)$.

We assume that:

- $\alpha(x_1, \ldots, x_n) = \beta(y_1, \ldots, y_m) \Rightarrow \Gamma_1(x_1, \ldots, x_n, y_1, \ldots, y_m) \in \mathcal{R}_{\mathrm{lt}}$,
- $C_1 = (\,\vec{u} \mid \alpha(t_1, \ldots, t_n) = \beta(s_1, \ldots, s_m), \Gamma\,)$
  $\rightsquigarrow (\,\vec{u} \mid \widehat{\Gamma_1}(t_1, \ldots, t_n, s_1, \ldots, s_m), \Gamma\,) = C_2$ by Interaction rule,
- $\mathsf{ind2env}(C_1) = ((\,\vec{u'} \mid \alpha(t'_1, \ldots, t'_n) = \beta(s'_1, \ldots, s'_m), \Gamma'\,), \mathrm{E})$.

$\Gamma_1$ does not contain any indirection terms because $\Gamma_1$ is the RHS of the interaction rule, and thus the following holds:

$$\mathsf{ind2env}(C_2) = ((\,\vec{u'} \mid \widehat{\Gamma_1}(t'_1, \ldots, t'_n, s'_1, \ldots, s'_m), \Gamma'\,), \mathrm{E}).$$

Then,

$$\begin{aligned}\mathsf{ToState}(C_1) &= (\,\mathrm{E} \mid \vec{u'} \mid - \mid \alpha(t'_1, \ldots, t'_n) = \beta(s'_1, \ldots, s'_m), \Gamma'\,),\\ \mathsf{ToState}(C_2) &= (\,\mathrm{E} \mid \vec{u'} \mid - \mid \widehat{\Gamma_1}(t'_1, \ldots, t'_n, s'_1, \ldots, s'_m), \Gamma'\,),\end{aligned}$$

and thus $\mathsf{ToState}(C_1) \Longrightarrow \mathsf{ToState}(C_2)$ by the transition rule A.

Var1: We assume that:

- $C_1 = (\,\vec{u} \mid x = t, \Gamma\,) \rightsquigarrow (\,\vec{u}[\$t/x] \mid \Gamma[\$t/x]\,) = C_2$ by Var1,
- $\mathsf{ind2env}(C_1) = ((\,\vec{u'} \mid x = t', \Gamma'\,), \mathrm{E})$.

Since $x$ is fresh for $\vec{u}[\$t/x]$ and $\Gamma[\$t/x]$, we obtain $((\,\vec{u'} \mid \Gamma'\,), \{(x, t')\} \cup \mathrm{E})$ as the result of $\mathsf{ind2env}(C_2)$ by the assumption of $\mathsf{ind2env}(C_1)$. Then,

$$\begin{aligned}\mathsf{ToState}(C_1) &= (\,\mathrm{E} \mid \vec{u'} \mid - \mid x = t', \Gamma'\,),\\ \mathsf{ToState}(C_2) &= (\,\{(x, t')\} \cup \mathrm{E} \mid \vec{u'} \mid - \mid \Gamma'\,),\end{aligned}$$

and thus $\mathsf{ToState}(C_1) \Longrightarrow \mathsf{ToState}(C_2)$ by the transition rule B.1.

Indirection1: We assume that:

- $C_1 = (\,\vec{u} \mid \$t = s, \Gamma\,) \rightsquigarrow (\,\vec{u} \mid t = s, \Gamma\,) = C_2$ by Indirection1,
- $\mathsf{ind2env}(\$t = s) = (x = s', \{(x, t')\} \cup \mathrm{E}_1 \cup \mathrm{E}_2)$ is obtained as follows:

$$\dfrac{\dfrac{\vdots}{\mathsf{ind2env}(t) = (t', \mathrm{E}_1)}}{\mathsf{ind2env}(\$t) = (x, \{(x, t')\} \cup \mathrm{E}_1)}\,\mathit{indirection} \qquad \dfrac{\vdots}{\mathsf{ind2env}(s) = (s', \mathrm{E}_2)}}{\mathsf{ind2env}(\$t = s) = (x = s', \{(x, t')\} \cup \mathrm{E}_1 \cup \mathrm{E}_2)}\,\mathit{equation}$$

- $\mathsf{ind2env}(C_1) = ((\,\vec{u'} \mid x = s', \Gamma'\,), \{(x, t')\} \cup \mathrm{E})$ where $\mathrm{E} \supseteq \mathrm{E}_1 \cup \mathrm{E}_2$.

Then, $\mathsf{ind2env}(t = s) = (t' = s', \mathrm{E}_1 \cup \mathrm{E}_2)$ by the assumption, and thus $\mathsf{ind2env}(C_2)$ is obtained as $((\,\vec{u'} \mid t' = s', \Gamma'\,), \mathrm{E})$. Since

$$\begin{aligned}\mathsf{ToState}(C_1) &= (\,\{(x, t')\} \cup \mathrm{E} \mid \vec{u'} \mid - \mid x = s', \Gamma'\,),\\ \mathsf{ToState}(C_2) &= (\,\mathrm{E} \mid \vec{u'} \mid - \mid t' = s', \Gamma'\,),\end{aligned}$$

$\mathsf{ToState}(C_1) \Longrightarrow \mathsf{ToState}(C_2)$ by the transition rule C.1.

Var2 and Indirection2: Similar to the cases of Var1 and Indirection1, respectively. □

**Lemma 4.4.17**

*Let* $(\,\vec{u} \mid \Gamma\,)$ *be a* $\rightsquigarrow$*-normal form, then* $\Gamma$ *is empty.*

**Proof.** Assume that $\Gamma$ is not empty.

- When $\Gamma$ is $\alpha(\vec{t}) = \beta(\vec{u}), \Gamma_1$: By the assumption of this section such that for every active pair $\alpha(\vec{t}) = \beta(\vec{s})$ there is an interaction rule $\alpha(\vec{x}) = \beta(\vec{y}) \in \mathcal{R}_{\mathrm{lt}}$, the Interaction rule can be applied.
- When $\Gamma$ is $x = t, \Gamma_1$ where $t \neq \$s$: The rule Var1 can be applied.
- When $\Gamma$ is $t = x, \Gamma_1$ where $t \neq \$s$: The rule Var2 can be applied.
- When $\Gamma$ is $\$t = s, \Gamma_1$: The rule Indirection1 can be applied.
- When $\Gamma$ is $t = \$s, \Gamma_1$: The rule Indirection2 can be applied.

Thus $\Gamma$ must be empty. □

**Theorem 4.4.18 (Completeness for the simpler abstract machine)**

*Let* $C_1$ *and* $C_2$ *be configurations in the simpler textual calculus such that* $C_1 \Downarrow C_2$*. Then* $\mathsf{ToState}(C_1) \Downarrow \mathsf{ToState}(C_2)$.

**Proof.** By Lemma 4.4.16, $\mathsf{ToState}(C_1) \Longrightarrow^* \mathsf{ToState}(C_2)$. When we assume $C_2 = (\,\vec{u} \mid \Gamma\,)$, then $\Gamma$ is empty by Lemma 4.4.17. Since $\mathsf{ToState}(\,\vec{u} \mid\,)$ has no code, $\mathsf{ToState}(C_2)$ is a normal form. □

Finally we define an operation Update to obtain an interface as a calculation result. Substitution rules are performed in the course of computation, thus the operation of Update is to apply Collect rules and to remove indirection terms $\$t$.

**Definition 4.4.19**

- *We define an operation* remInd *for terms to replace an indirection term* $\$t$ *with* $t$ *as follows:*
  - $\mathsf{remInd}(x) \stackrel{\mathrm{def}}{=} x$,
  - $\mathsf{remInd}(\$t) \stackrel{\mathrm{def}}{=} \mathsf{remInd}(t)$,
  - $\mathsf{remInd}(\alpha(t_1, \ldots, t_n)) \stackrel{\mathrm{def}}{=} \alpha(\mathsf{remInd}(t_1), \ldots, \mathsf{remInd}(t_n))$.
- *We extend the operation* remInd *for sequences of terms:*
  - $\mathsf{remInd}(t_1, \ldots, t_n) \stackrel{\mathrm{def}}{=} \mathsf{remInd}(t_1), \ldots, \mathsf{remInd}(t_n)$.
- *We define an operation* Update *for a configuration* $(\,\vec{u} \mid \Gamma\,)$ *as follows:*

– $\mathsf{Update}(\,\vec{u} \mid x = t, \Gamma\,) \stackrel{\text{def}}{=} \mathsf{Update}(\,\vec{u}[t/x] \mid \Gamma[t/x]\,)$,

– $\mathsf{Update}(\,\vec{u} \mid t = x, \Gamma\,) \stackrel{\text{def}}{=} \mathsf{Update}(\,\vec{u}[t/x] \mid \Gamma[t/x]\,)$,

– $\mathsf{Update}(\,\vec{u} \mid -\,) \stackrel{\text{def}}{=} \mathsf{remInd}(\vec{u})$.

**Example 4.4.20**

*The computation of the configuration* $(\,r \mid \mathtt{Add}(r, \mathtt{Z}) = \mathtt{S}(\mathtt{Z})\,)$ *in Figure 2.1 is given below:*

$(\,r \mid \mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(\mathtt{Z})\,)$

$\rightsquigarrow (\,r \mid \mathtt{Add}(\mathtt{Z}, x) = \mathtt{Z}, r = \mathtt{S}(x)\,)$ (Interaction)

$\rightsquigarrow (\,r \mid \mathtt{Z} = x, r = \mathtt{S}(x)\,)$ (Interaction)

$\rightsquigarrow (\,r \mid r = \mathtt{S}(\$\mathtt{Z})\,)$ (Var2)

$\rightsquigarrow (\,\$\mathtt{S}(\$\mathtt{Z}) \mid\,)$. (Var1)

$\mathsf{Update}(\,\$(\mathtt{S}(\$\mathtt{Z})) \mid\,)$

$= \mathsf{remInd}(\$(\mathtt{S}(\$\mathtt{Z})))$

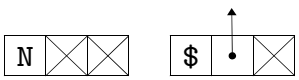
$= \mathtt{S}(\mathtt{Z})$.

## 4.5 Encoding method

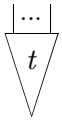
In this section we propose an encoding method of the simpler textual calculus, which is called a *single link encoding method* based on the standardised implementation model.

### 4.5.1 Implementation model

This abstract machine of the textual calculus is a refinement of the lightweight abstract machine, and thus the encoding method for agents is the same as the Directed one. Names and indirection are represented as graph nodes whose ids are `ID_NAME` and `ID_INDIRECTION` and arities 0 and 1 respectively. Name and indirection nodes are drawn as the following nodes `N` and `$`, when we fix the maximum arity as 2 for example:




In addition, the consistency assures that there are no mutual connections such that $\mathrm{E}[x] = y$ and $\mathrm{E}[y] = x$, therefore every net has a tree-like data-structure and we draw a net for a term $t$ as follows:



where the bottom is a principal port of agent nodes and the top ports are free ports of the net.

With respect to the transitions in Figure 4.1, the rule A is the same as Directed encoding method, and the others are drawn as Figure 4.2. Thus, the run-time function `eval` is written as follows:

```
void eval() {
  Agent *a1, *a2;
  while (popActive(&a1, &a2)) {
    if (a2->id != ID_NAME) {
      if (a1->id != ID_NAME) {
        R[a1->id][a2->id](a1, a2);
      } else if (a1->id == ID_INDIRECTION) {
        /* C.1 */
        Agent *a1p0 = a1->port[0];
        freeAgent(a1);
        pushActive(a1p0, a2);
      } else {
        /* B.1 */
        a1->port[0] = a2;
        a1->id = ID_INDIRECTION;
      }
    } else if (a2->id == ID_INDIRECTION) {
      /* C.2 */
      Agent *a2p0 = a2->port[0];
      freeAgent(a2);
      pushActive(a1, a2p0);
    } else {
      /* B.2 */
      a2->id = ID_INDIRECTION;
      a2->port[0] = a1;
    }
  }
}
```

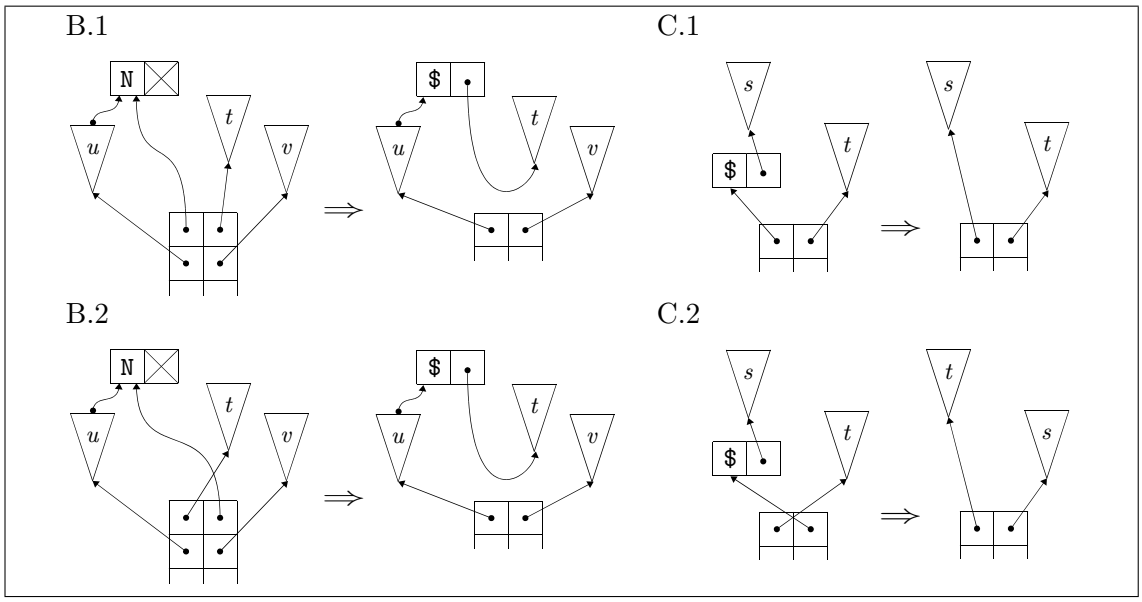



Figure 4.2: Transition rules in Figure 4.1

For instance, the net in Figure 2.1 is evaluated by this `eval` function as shown in Figure 4.3.

### 4.5.2 Reduction strategies

In this section we show how some reduction strategies for equations can be realised in this implementation. Here, to generalise a data structure for equations, we use a list structure as follows:

```
typedef struct ApList {
  struct Active ap;
  struct ApList *next;
} ApList;

ApList *newApList(Agent *a1, Agent *a2, ApList *next) {
  ApList *alist = malloc(sizeof(ApList));
  alist->ap.a1 = a1;
  alist->ap.a2 = a2;
  alist->next = next;
  return alist;
}

void freeApList(ApList *alist) {
```

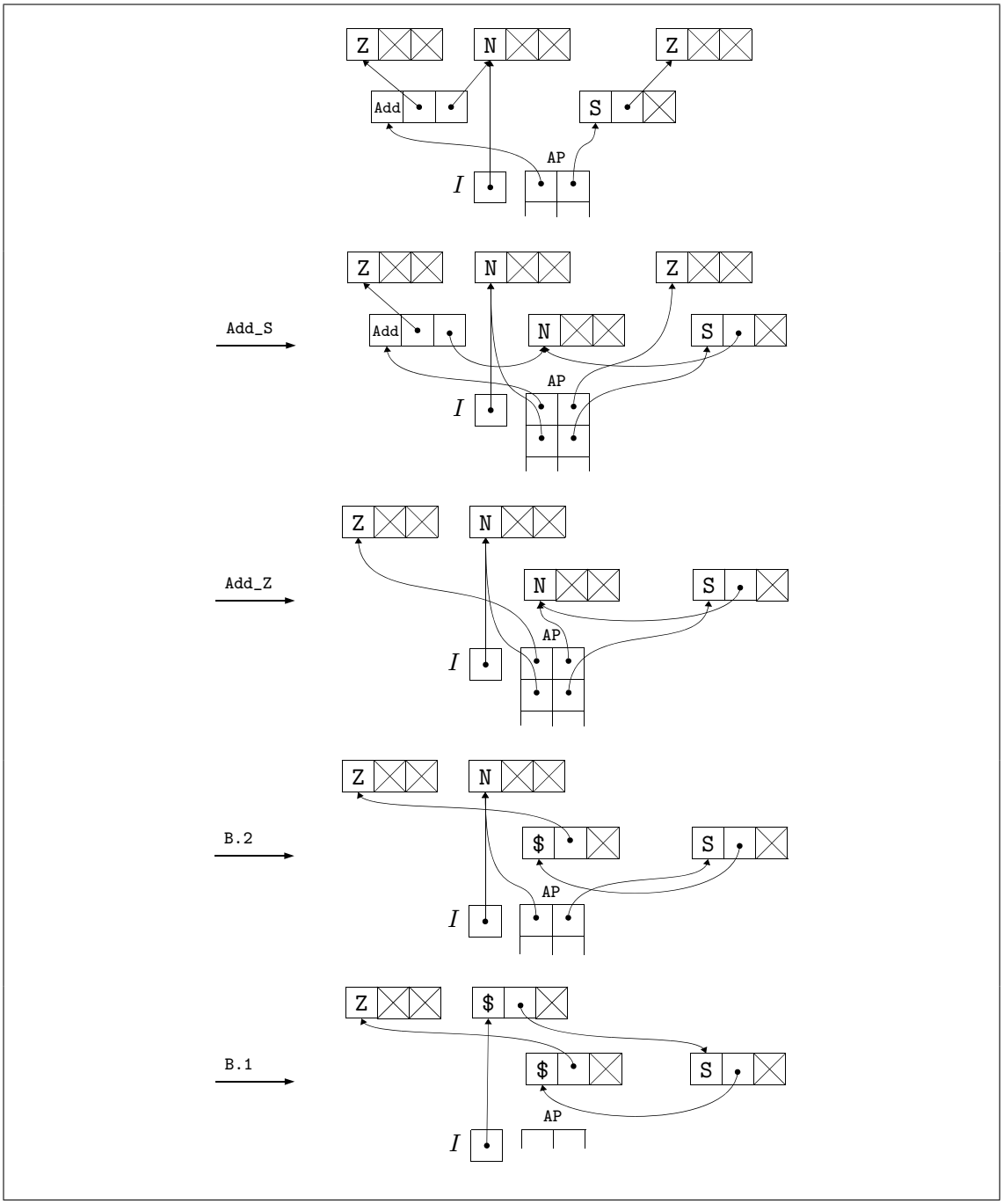
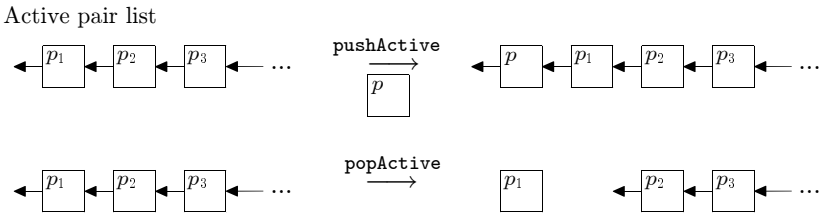



Figure 4.3: Single link encoding method: evaluation of the net in Figure 2.1

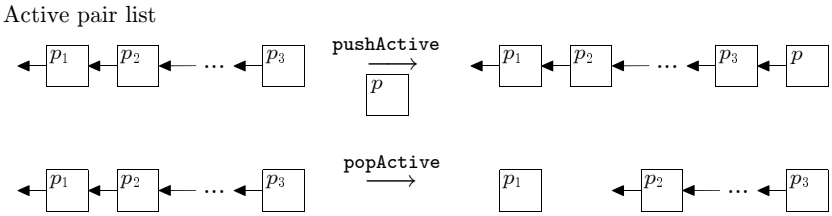
```
  free(alist);
}
```

**LIFO (Last In, First Out)** LIFO stands for "Last In, First Out", and in the implementation this means that an equation created last is operated first, thus equations in some specific area tend to be performed eagerly. This is typically realised by a stack. Here we implement such a stack by using the list data structure as follows:




```
/* to manipulate the equation stack */
ApList *Aps = NULL;

/* Implementation of functions for active pairs*/
void pushActive(Agent *a1, Agent *a2) {
  Aps = newApList(a1, a2, Aps);
}
int popActive(Agent **a1, Agent **a2) {
  if (Aps == NULL) return 0;
  *a1 = Aps->ap.a1;
  *a2 = Aps->ap.a2;
  ApList *tmp = Aps->next;
  freeApList(Aps);
  Aps = tmp;
  return 1;
}
```

**FIFO (First In, First Out)** FIFO stands for "First In, First out", and in the implementation this means that equations are operated in the order created, thus every equation is performed evenly. This is typically realised by a queue, and implemented by using the list data structure as follows:




```
/* to manipulate the equation stack */
ApList *Top = NULL;
ApList *Bottom = NULL;

/* Implementation of functions for active pairs*/
void pushActive(Agent *a1, Agent *a2) {
  ApList *newlist = newApList(a1, a2, NULL);
  if (Top == NULL) {
    Top = newlist;
    Bottom = newlist;
  } else {
    Bottom->next = newlist;
    Bottom = newlist;
  }
}
int popActive(Agent **a1, Agent **a2) {
  if (Top == NULL) {
    Bottom = NULL;
    return 0;
  }
  *a1 = Top->ap.a1;
  *a2 = Top->ap.a2;
  ApList *tmp = Top->next;
  freeApList(Top);
  Top = tmp;
  return 1;
}
```

**Weak reduction** Weak reduction strategy (Definition 2.1.12) evaluates only equations that contain names of the interface. First, we introduce two functions in order to check if

the given term includes those of the interface:

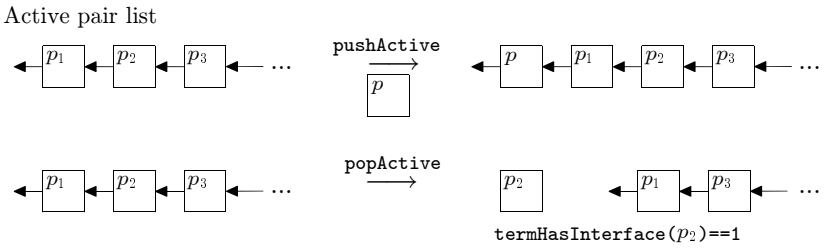
- `nameInInterface` takes a pointer to an agent node and returns `1` if there is the same pointer in the interface list. Otherwise, it returns `0`.

```
int nameInInterface(Agent *a) {
  int i;
  for (i=0; i<SIZE_INTERFACE) {
    if (a == I[i]) return 1;
  }
  return 0;
}
```

- `termHasInterface` takes a pointer to an agent node and returns `1` if it contains one of the interface. Otherwise, it returns `0`.

```
int termHasInterface(Agent *a) {
  int i;
  for (i=0; i<SIZE_INTERFACE) {
    if (a->id == ID_NAME) {
      return nameInInterface(a);
    } else {
      int j;
      for(j=0; j < Arities[a->id]; j++) {
        if (termHasInterface(a->port[j])) {
          return 1;
        }
      }
      return 0;
    }
  }
}
```

By using those functions, the function `pushActive` and `popActive` are coded as follows:




```
/* to manipulate the equation stack */
ApList *Aps = NULL;

/* Implementation of functions for active pairs*/
void pushActive(Agent *a1, Agent *a2) {
  Aps = newApList(a1, a2, Aps);
}
int popActive(Agent **a1, Agent **a2) {
  ApList *before, *alist;
  alist = Aps;
  while (alist != NULL) {
    if (termHasInterface(alist->ap.a1)
     || termHasInterface(alist->ap.a2)) {
      *a1 = alist->ap.a1;
      *a2 = alist->ap.a2;
      if (alist == Aps) {
        Aps = alist->next;
      } else {
        before->next = alist->next;
      }
      freeApList(alist);
      return 1;
    }
    before = alist;
    alist = alist->next;
  }
  return 0;
}
```

| | Undirected(INET) | Undirected($\mathsf{in}^2$) | Directed(Light) | Directed(Single) |
|---|---|---|---|---|
| $F_{32}$ | 1.58 | 1.37 | 1.52 | 1.49 |
| $F_{33}$ | 2.62 | 2.29 | 2.52 | 2.49 |
| $F_{34}$ | 4.37 | 3.80 | 4.21 | 4.15 |
| $A(3, 10)$ | 1.77 | 1.42 | 1.59 | 1.58 |
| $A(3, 11)$ | 7.12 | 5.73 | 6.44 | 6.39 |
| $A(3, 12)$ | 29.47 | 24.01 | 26.39 | 26.14 |
| `2 7 6 I I` | 0.73 | 0.71 | 1.26 | 1.28 |
| `2 7 7 I I` | 2.12 | 2.13 | 3.58 | 3.68 |

Table 4.1: The execution time in seconds on the standardised implementation model

### 4.5.3 Experimental results

In this section we examine the execution time in the single link encoding methods for programs in Section 3.4.3, compared to other methods.

Table 4.1 is obtained by adding the execution time of the single link encoding method (denoted as "Single") into Table 3.1. The trend in the execution time is almost the same as the encoding method of amineLight, and thus in terms of the cost, Undirected encoding method of $\mathsf{in}^2$ is the best, though the single link encoding method is not the worst.

In comparison with the amineLight encoding method, the single link encoding method computes Fibonacci number and Ackermann function a little faster. On the other hand, Application of Church numerals demands a lot of computation for names, especially equations such as $x = y$, and those operations take a little more time than the amineLight encoding. This can be caused, as shown in Section 4.3.2, by the absence of a map for connections between names in the single link encoding method, while the amineLight encoding uses the map. Table 4.2 shows ratios of name operations to interaction operations. With respect to the computation of Application of Church numerals, it increases to 4.12, whereas it is 2.75 in Directed encoding method as shown in Table 3.2. Even though the cost of each operation for those names is quite small, as shown in the computation of Fibonacci numbers that is faster where the ratio increases by 0.18, the significant accumulation of the cost leads to the less efficient system. Thus, it is important to reduce the cost of operations for names. By changing the data structure it can be improved to some extent. Further discussion will be made about this topic in Section 7.1.1.

In the single link encoding method, as shown in Section 4.3.2, only a single side of an equation is managed, while in the amineLight encoding both sides of an equation such as

| | Interactions | Names | Names/Interactions |
|---|---|---|---|
| $F_{32}$ | 74636718 | 65106325 | 0.87 |
| $F_{33}$ | 123315177 | 105344341 | 0.85 |
| $F_{34}$ | 203654818 | 170450820 | 0.84 |
| $A(3,10)$ | 134103148 | 134094952 | 1.00 |
| $A(3,11)$ | 536641652 | 536625264 | 1.00 |
| $A(3,12)$ | 2147025020 | 2146992248 | 1.00 |
| `2 7 6 I I` | 15676873 | 64538288 | 4.12 |
| `2 7 7 I I` | 46118916 | 190190039 | 4.12 |

Table 4.2: The number of operations in the single link method

$x = y$ are managed using the map. When we take parallel execution into account, this takes advantage of the locality of the rewriting. The critical sections are caused only by B.1 and B.2 since the name agents can be linked by two active pairs as shown in Figure 4.2. Moreover, those are performed by connecting the ports of names into other principal ports of unlocked agent nodes, therefore these can be locked with an atomic operation like CAS (Compare-and-swapping). For instance, when we represent name and indirection nodes as just name nodes, we can regard name nodes whose ports are not NULL as indirection nodes. In the following picture, the left graph node is a name and the right one is an indirection:


Thus, the run-time function `eval` is written as follows:

```
#define CAS_SLEEP 4
void eval() {
  Agent *a1, *a2;
  while (popActive(&a1, &a2)) {
loop:
    if (a2->id != ID_NAME) {
      if (a1->id != ID_NAME) {
        R[a1->id][a2->id](a1, a2);
      } else if (a1->port[0] != NULL) {
        /* C.1 */
        Agent *a1p0 = a1->port[0];
```

```
            freeAgent(a1);
            pushActive(a1p0, a2);
          } else {
            /* B.1 */
            if (!(__sync_bool_compare_and_swap(
                   &(a1->port[0]), NULL, a2))) {
              usleep(CAS_USLEEP); // wait a little
              goto loop;          // and retry
            }
          }
        } else if (a2->port[0] != NULL) {
          /* C.2 */
          Agent *a2p0 = a2->port[0];
          freeAgent(a2);
          pushActive(a1, a2p0);
        } else {
          /* B.2 */
          if (!(__sync_bool_compare_and_swap(
                 &(a2->port[0]), NULL, a1))) {
            usleep(CAS_USLEEP); // wait a little
            goto loop;          // and retry
          }
        }
      }
    }
```

Generally, in parallel execution, we have to manage other critical sections that arise in the allocation and deallocation of graph nodes, and in the operations of the active pair stack. Moreover, multi-thread executions require synchronisation so that those threads can be controlled well. We will discuss those mechanism when we introduce a multi-threaded interpreter in Section 7.1.2, and compare performance to other evaluators of interaction nets.

## 4.6 Summary

In this chapter, we proposed a new method of encoding interaction nets, which is a refinement of the method used in amineLight. This method requires only single links, and thus every net is encoded as a tree-like data-structure. Moreover, this can be used to derive parallel execution models naturally.

We also introduced the new abstract machine and a textual calculus, and an encoding method based on the standardised implementation model.

Finally, we gave a comparison with other encoding methods that shows not the best in terms of efficiency, thus at most about 1.8 times slower than the best, but it can be recovered by parallel execution. This is discussed in Section 7.1.2 again.

# Chapter 5

# Low-level language LL0

In this chapter we propose the low-level language LL0 which defines a low level set of instructions that can build a net and reduce it to normal form. Specific instructions in LL0 have an almost one-to-one correspondence to the standardised implementation model in the single encoding method. In addition, these instructions are considered as byte-codes of an abstract machine.

First, we define LL0 and we give a compilation from interaction nets into LL0 instructions. Next, we illustrate how LL0 corresponds to the standardised implementation model and finally, we show how LL0 can be used as an instruction set for an abstract machine.

## 5.1 The Low-level language LL0

In this section we introduce a low-level language to implement interaction nets. The concrete representation of a configuration can be summarised by a diagram in Figure 5.1, where

- $\Gamma$ represents heaps of graph elements for a net,
- $EQ$ a stack of equations,
- $I$ an interface

and interaction rules are represented by:

- a set of procedures to perform interaction rules.

In the next section we show how to construct nets and in the following section we illustrate how to represent interaction rules.

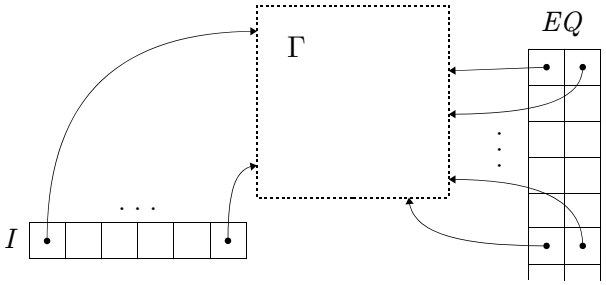



Figure 5.1: Configuration

### 5.1.1 Constructing nets

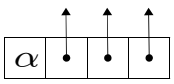
For a net, there are three kinds of graph element: agents, names and indirection nodes. Each of these elements is allocated memory in the heap. An element, such as an agent, may contain pointers to other elements (to represent auxiliary ports). Intuitively, an agent can be represented in the C language as follows:

```
typedef struct Agent {
  int id;
  struct Agent *port[];
} Agent;
```

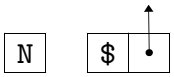
In this `Agent` structure, each symbol $\alpha_1, \ldots, \alpha_n$ for agents is distinguished by a unique number `id`. The number of the `port` $p$, which is called arity, corresponds to a number of auxiliary ports of an agent. To assign an arity to an agent, we use the following declaration `#agent`:

`#agent` $\alpha_1 : p_1, \ldots, \alpha_n : p_n$

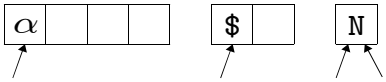
where $p_i$ is an arity for an agent symbol $\alpha_i$ such that $ar(\alpha_i) = p_i$. After this declaration, these symbols $\alpha_i$ are recognised as unique numbers and those arities $p_i$ can be referred by a function arity such that $\mathsf{arity}(\alpha_i) = p_i$. We draw an agent node whose `id` is the number assigned for the symbol $\alpha$ and arity is 3 as follows:




Name and indirection nodes are graph elements whose `id`s are denoted by symbols `N` and `$`, and arities are 0 and 1 respectively, where `N` and `$` are drawn from a set that does not overlap with the set of agent symbols. We draw these nodes as follows:




Agents and indirection nodes can point to other graph elements; variables represent leaves of the structure and point no further. Agents and indirection nodes can be pointed to from at most one of other elements while name nodes can be pointed from two other elements.




To allocate an agent graph element whose id is *id*, we use the following instruction:

$$\texttt{x} = \texttt{mkAgent(}id\texttt{)}$$

This instruction sets a pointer of the allocated memory to the variable `x` used by the rest of the machinery.

For instance, by the following code, a term `Z` is assigned to a variable `aZ`:

```
/* Definition of Agents */
#agent Z:0


/* Z */
aZ=mkAgent(Z)
```

The result of executing the above two instructions is an agent node `Z` (with arity 0 and) with no pointers:


To dispose of an allocation $a$ of a graph element, we use an instruction `free(`$a$`)`. For instance, the following code disposes of the above allocation of the `aZ`:

```
free(aZ)
```

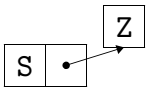
A connection between a principal port and an auxiliary port is encoded by an assignment. In this language, to assign a pointer of an existing graph element $b$ to a port $p$ of another graph element $a$, we use the following instruction:

$$a\texttt{[}p\texttt{]=}b$$

We note that the index of these ports starts from 1. For instance, a term `S(Z)` is encoded as follows:

```
/* Definition of Agents */
#agent Z:0, S:1

/* S(Z) */
aS=mkAgent(S)
aZ=mkAgent(Z)
aS[1]=aZ
```

This connection can be represented graphically as follows:




We restrict that only one port can refer to an agent node.

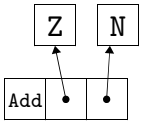
To allocate a name node, we use the following instruction:

```
x=mkName()
```

For instance, a term `Add(Z,r)` is encoded as follows:

```
/* Definition of Agents */
#agent Z:0, Add:2

/* Add(Z,r) */
aAdd=mkAgent(Add)
aZ=mkAgent(Z)
r=mkName()
aAdd[1]=aZ
aAdd[2]=r
```

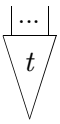
and depicted as follows:




To allocate an indirection one, we use the following instruction:

```
x=mkInd()
```

Generally, when agent nodes are connected together, they are trees that we represent in the following way, where the free names are at the top of the tree:



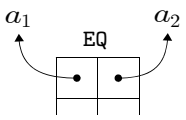

Next, we introduce a stack of equations `EQ`. The stack is initially created empty. Intuitively, an element of this stack can be written using the following code fragment in the C language:

```
typedef struct Equation {
  Agent *a1;
  Agent *a2;
} Equation;
```

An equation node can point to two graph elements. To create an equation of two graph elements $a_1, a_2$ in the stack `EQ`, we use the following instruction:

`push(`$a_1$`,`$a_2$`)`

We draw this as follows:




To pop an equation from the top of the stack `EQ`, we use the following instruction:

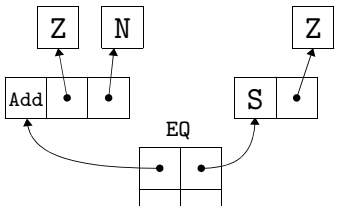
```
stackFree()
```

Like in the textual calculus, we represent a connection between principal ports by creating an equation between the two agent nodes into the stack. For instance, we can encode an equation `Add(Z,`$r$`) = S(Z)` in the configuration $\langle\ r\ \mid$ `Add(Z,`$r$`) = S(Z)` $\rangle$ in Figure 2.1 as follows:

```
/* Definition of Agents */
#agent Z:0, S:1, Add:2


/* Add(Z,r) */
aZ=mkAgent(Z)
aAdd=mkAgent(Add)
r=mkName()
aAdd[1]=aZ
aAdd[2]=r
```

```
/* S(Z) */
bS=mkAgent(S)
bZ=mkAgent(Z)
bS[1]=bZ

/* Add(Z,r)=S(Z) */
push(aAdd,bS)
```

and we depict it as follows:




Next, we introduce the interface `I`. Interfaces are of fixed size $n$ as the size of the observable interface of a net can be pre-determined (and it is preserved during execution). Interfaces are created with the following instruction:

```
I = mkInterface(n)
```

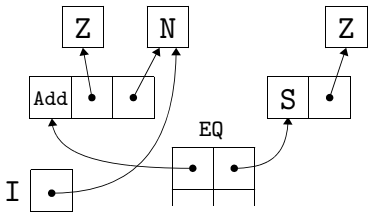
Those elements are accessed using the usual array notation `I[1]`, ... , `I[n]`, and can point to one graph element.

For instance, the configuration $\langle\ r \mid \mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(\mathtt{Z})\ \rangle$ is encoded as follows:

```
/* Definition of Agents */
#agent Z:0, S:1, Add:2

/* create the interface */
I = mkInterface(1)

/* Add(Z,r) */
aZ=mkAgent(Z)
aAdd=mkAgent(Add)
r=mkName()
aAdd[1]=aZ
```

```
aAdd[2]=r

/* S(Z) */
bS=mkAgent(S)
bZ=mkAgent(Z)
bS[1]=bZ

/* Add(Z,r)=S(Z) */
push(aAdd,bS)

/* set the interface */
I[1]=r
```

and represented with each name of the interface pointing to a corresponding name node as follows:

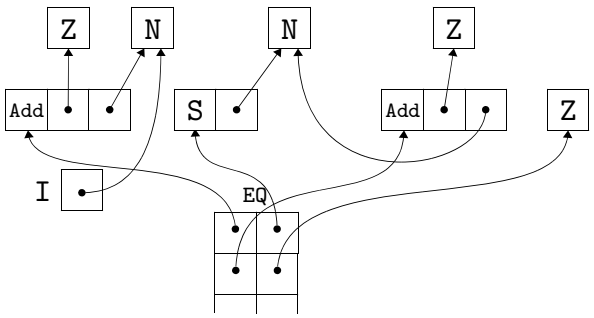



For a connection between two auxiliary ports, we assign one name node to two ports. For instance, the configuration $\langle\ r\ \mid\ \mathtt{Add}(\mathtt{Z}, r) = \mathtt{S}(w), \mathtt{Add}(\mathtt{Z}, w) = \mathtt{Z}\ \rangle$ in Figure 2.2 is encoded as follows:

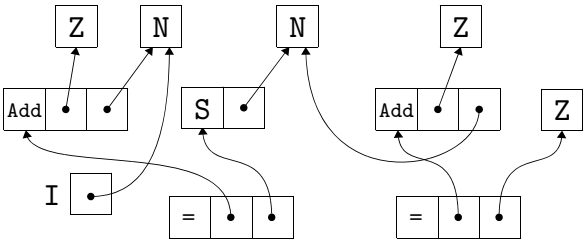
```
/* Definition of Agents */
#agent Z:0, S:1, Add:2

/* create the interface */
I = mkInterface(1)

/* Add(Z,r) */
aZ=mkAgent(Z)
aAdd=mkAgent(Add)
r=mkName()
aAdd[1]=aZ
aAdd[2]=r
```

```
/* S(w) */
bS=mkAgent(S)
w=mkName()
bS[1]=w

/* Add(Z,r)=S(w) */
push(aAdd,bS)

/* Add(Z,w) */
aZ=mkAgent(Z)
aAdd=mkAgent(Add)
aAdd[1]=aZ
aAdd[2]=w

/* Z */
bZ=mkAgent(Z)

/* Add(Z,w)=Z */
push(aAdd,bZ)

/* set the interface */
I[1]=r
```

and depicted as follows:




To avoid complex wiring, we introduce the following notation for equation nodes, and we assume that our equation stack is drawn horizontally with the top placed at the most left side.




### 5.1.2 Defining interaction rules

Next, we introduce *rule procedures* to perform interaction rules. For an interaction rule between $\alpha(\vec{x})$ and $\beta(\vec{y})$, we describe a rule procedure as follows:

```
rule α β {
  ⋮
}
```

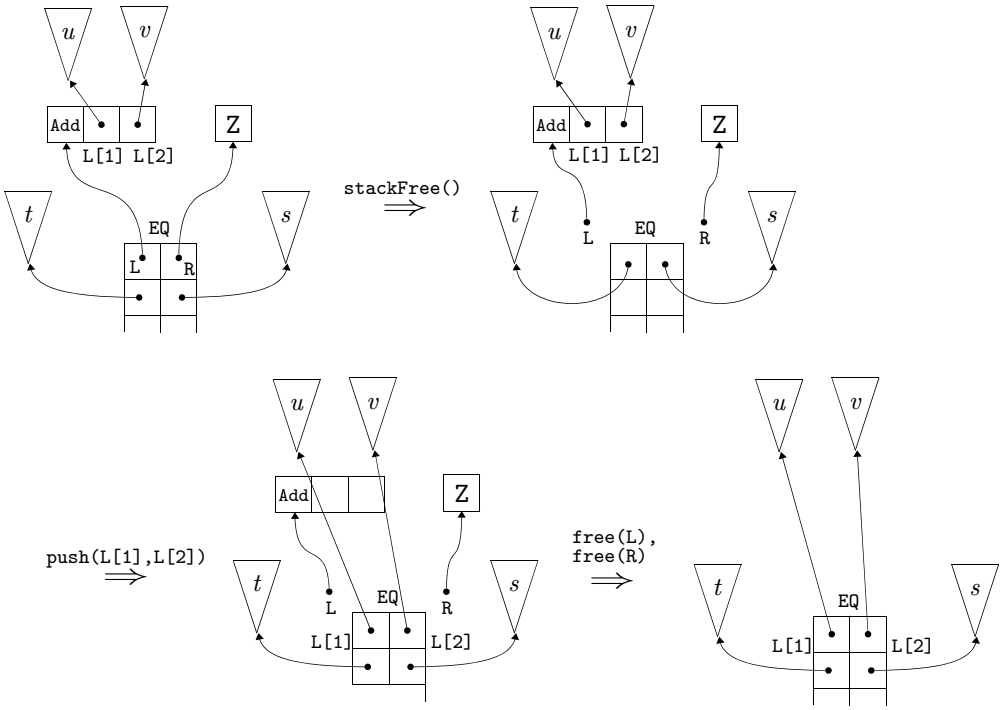
We can write instructions between { and }, which we call a rule block, and the scope of variables is within a rule block. In execution, the procedures provide special variables `L`, `R` that are pointers to the left and the right hand side agents of the active pair equation. For instance, for a rule $\texttt{Add}(x_1, x_2) = \texttt{Z} \Rightarrow x_1 = x_2$ in Figure 2.1, the function can be written as follows:

```
rule Add Z {
  stackFree()
  push(L[1],L[2])
  free(L)
  free(R)
}
```

where the variables `L` and `R` can be used as pointers to the active pair agents `Add` and `Z` respectively, and the arguments $x_1$ and $x_2$ can be pointed by `L[1]` and `L[2]` respectively.

In the rule procedure, we can also write an instruction `stackFree()` to remove the top of the equation stack. The elements of the current top equation can be overwritten with special variables `StackL` and `StackR` that contain pointers to the LHS and RHS in the equation on the top of the stack respectively.

These rule procedures are represented as transformations on the data structure. For instance, the rule between `Add` and `Z` is depicted with labels `L`, `R`, `L[1]` and `L[2]` as follows:




To give an example for the other rule $\texttt{Add}(x_1, x_2) = \texttt{S}(y_1) \Rightarrow \texttt{Add}(x_1, w) = y_1, x_2 = \texttt{S}(w)$ in Figure 2.1, the procedure creates new nodes of `S`, `Add` and `w`, and re-wires each port according to the RHS in the rule. This can be written as follows:

```
rule Add S {
  stackFree()

  aS=mkAgent(S)
  aAdd=mkAgent(Add)
  w=mkName()

  aAdd[1]=L[1]
  aAdd[2]=w
  push(aAdd, R[1])

  aS[1]=w
  push(L[2], aS)

  free(L)
  free(R)
}
```

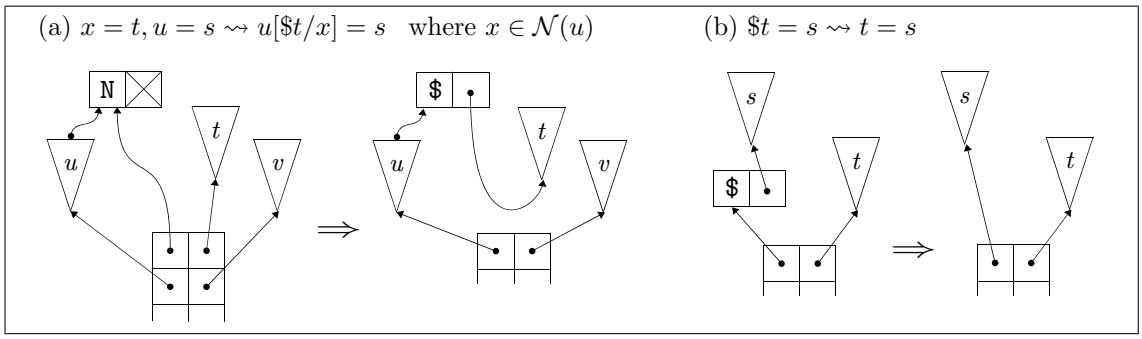



Figure 5.2: computation rules for name and indirection nodes

Figure 5.2 (a) and (b) are instances of Var1 and Indirection1 rules to illustrate the ideas.

### 5.1.3 Instructions and Syntax of LL0

In this section, we define the syntax and instructions in LL0.

Figure 5.3 summarises the instruction set of LL0. The port numbers start from 1, and by using the instruction $x$`[`$p$`]=`$y$, we can assign a graph node $y$ to a port $p > 0$ of a graph node $x$. We also use the port 0 to refer to the id element. For instance, $x$`[0]=`$\alpha$ changes the id of an agent node $x$ into $\alpha$.

We define the syntax of LL0 as shown in Figure 5.4 where ALPHA means one of letters `a...zA...Z` and DIGIT means one of letters `0...9`. Instructions in Figure 5.3 and rule procedures using the instructions are accepted by $\langle$*instruction*$\rangle$ and $\langle$*defRule*$\rangle$ in Figure 5.4, respectively.

## 5.2 Translation of the textual calculus into LL0

In this section, we introduce a translation of the simpler text calculus into LL0.

We use a set of pairs and operations for the pairs defined in Definition 3.3.11. We also use the following notations for strings:

**Definition 5.2.1 (Notations for strings)**

- *We use* Str *as a set of strings.*
- *We use “ and ” as a pair of delimiters to represent a string explicitly. For instance, we write a string* `abc` *as “*`abc`*”.*
- *We use the notation* $\{x\}$ *in a string as the result of replacing the occurrence* $\{x\}$ *with its actual value. For instance, if* $x =$ “`abc`” *and* $y = 89$ *then “*`1`$\{x\}$`2`$\{y\}$*” =*

| Instruction | Description |
|---|---|
| `#agent` $\alpha_1 : p_1, \ldots, \alpha_n : p_n$ | Declare $\alpha_1, \ldots, \alpha_n$ as symbols of agents whose arity are $p_1, \ldots, p_n$. |
| `I=mkInterface(`$n$`)` | Create a fixed $n$-size interface and assign its pointer to the variable `I`. |
| $x$ `= mkAgent(`$id$`)` | Allocate (unused) memory for an agent node whose id is $id$ and assign it to the variable $x$. |
| $x$ `= mkName()` | Allocate (unused) memory for a name node, and assign it to the variable $x$. |
| $x$ `= mkInd()` | Allocate (unused) memory for an indirection node, and assign it to the variable $x$. |
| `free(`$x$`)` | Dispose of an assigned allocation $x$ of a graph element. |
| $x$`[`$p$`]=`$y$ | Assign a graph element $y$ to a port $p > 0$ of an agent node $x$. |
| $x$`[0]=`$\alpha$ | Change the id of an agent node $x$ into $\alpha$. |
| `push(`$x$`,`$y$`)` | Create an equation of two graph element $x, y$ in the stack of equations. |
| `stackFree()` | Dispose of an operating active pair in the rule procedure from the stack of equations. |

Figure 5.3: Instructions of LL0

"`1abc289`".

- *We use* + *as an infix binary operation to concatenate each string. For instance, if* $x =$ "`abc`"*, then* $x +$ "`123`" = "*abc*123".

### 5.2.1 Translation of configurations

A configuration $\langle\ \vec{u} \mid \Delta\ \rangle$ will be translated into the following structure:

**Definition 5.2.2 (Compilation of terms and nets)**

- *We use a set* $\mathrm{N} : \mathcal{N} \times \mathrm{Str}$*, which is called a* name table*, so that a name* $x \in \mathcal{N}$ *can correspond to a string of a variable name in a code sequence and those corresponding can be looked up from compilation functions. We define a function* `makeN` *to make such a name table and a code sequence for those names by a given*

$$
\begin{array}{rcl}
\langle \textit{instruction} \rangle & ::= & \langle \textit{declaration} \rangle \mid \langle \textit{operation} \rangle \\
\langle \textit{declaration} \rangle & ::= & \langle \textit{decAgent} \rangle \mid \langle \textit{decInterface} \rangle \\
\langle \textit{decAgent} \rangle & ::= & \texttt{'\#agent'}\ \langle \textit{agentArity} \rangle\ (\texttt{','}\ \langle \textit{agentArity} \rangle)^* \\
\langle \textit{agentArity} \rangle & ::= & \langle \textit{symbol} \rangle\ \texttt{':'}\ \langle \textit{num} \rangle \\
\langle \textit{decInterface} \rangle & ::= & \texttt{'I=mkInterface'}\ \texttt{'('}\ \langle \textit{num} \rangle\ \texttt{')'} \\
\langle \textit{operation} \rangle & ::= & \langle \textit{assignment} \rangle \mid \langle \textit{disposeAgent} \rangle \mid \langle \textit{opEquation} \rangle \\
\langle \textit{assignment} \rangle & ::= & \langle \textit{nodeExp} \rangle\ \texttt{'='}\ (\langle \textit{nodeExp} \rangle \mid \langle \textit{mkGraphElement} \rangle) \\
\langle \textit{nodeExp} \rangle & ::= & \langle \textit{var} \rangle \mid \langle \textit{var} \rangle\ \texttt{'['}\ \langle \textit{num} \rangle\ \texttt{']'} \\
\langle \textit{mkGraphElement} \rangle & ::= & \texttt{'mkAgent'}\ \texttt{'('}\ \langle \textit{symbol} \rangle\ \texttt{')'}) \mid \texttt{'mkName()'} \mid \texttt{'mkInd()'} \\
\langle \textit{disposeAgent} \rangle & ::= & \texttt{'free'}\ \texttt{'('}\ \langle \textit{nodeExp} \rangle\ \texttt{')'} \\
\langle \textit{opEquation} \rangle & ::= & \texttt{'push'}\ \texttt{'('}\ \langle \textit{nodeExp} \rangle\ \texttt{','}\ \langle \textit{nodeExp} \rangle\ \texttt{')'} \mid \texttt{'stackFree()'} \\
\langle \textit{symbol} \rangle & ::= & \text{ALPHA}\ \langle \textit{letter} \rangle^* \\
\langle \textit{var} \rangle & ::= & (\texttt{'\_'})^*\ \text{ALPHA}\ \langle \textit{letter} \rangle^* \\
\langle \textit{letter} \rangle & ::= & (\text{ALPHA} \mid \text{DIGIT} \mid \texttt{'\_'} \mid \texttt{''})^+ \\
\langle \textit{num} \rangle & ::= & (\text{DIGIT})^+ \\
 & & \\
\langle \textit{defRule} \rangle & ::= & \texttt{'rule'}\ \langle \textit{symbol} \rangle\ \langle \textit{symbol} \rangle\ \langle \textit{ruleBlock} \rangle \\
\langle \textit{ruleBlock} \rangle & ::= & \texttt{'\{'}\ \langle \textit{operation} \rangle^*\ \texttt{'\}'}
\end{array}
$$

Figure 5.4: Syntax of LL0

*name set* $\{x_1, \ldots, x_n\}$ *as follows:*

$$
\begin{array}{lcl}
\mathsf{makeN}(\{x_1, \ldots, x_n\}) & \stackrel{\text{def}}{=} & \mathsf{makeN'}(\{x_1, \ldots, x_n\}, \emptyset) \\
\mathsf{makeN'}(\{x_1, \ldots, x_n\}, \mathrm{N}) & \stackrel{\text{def}}{=} & \mathsf{let} \\
 & & \quad \mathrm{N}_0 = \mathrm{N}; \\
 & & \quad \mathsf{for}(1 \leq i \leq n): \\
 & & \quad\quad a_i = \mathsf{freshStr}(); \\
 & & \quad\quad c_i = \text{“}\{a_i\}\texttt{=mkName()}\text{”}; \\
 & & \quad\quad \mathrm{N}_i = (\mathrm{N}_{i-1}[x_i] := a_i); \\
 & & \mathsf{in} \\
 & & \quad (\mathrm{N}_n, c_1 + \cdots + c_n) \\
 & & \mathsf{end};
\end{array}
$$

- *We define a translation* $\mathsf{Compile}_s$ *from a symbol set* $\Sigma$ *into a code string as follows:*

$$
\begin{array}{lcl}
\mathsf{Compile}_s(\emptyset) & \stackrel{\text{def}}{=} & \text{“”} \\
\mid \mathsf{Compile}_s(\{\alpha_1, \ldots, \alpha_n\}) & \stackrel{\text{def}}{=} & \text{“\texttt{\#agent }”} + \\
 & & \text{“}\{\alpha_1\}\texttt{:}\{ar(\alpha_1)\}\text{”} + \cdots + \text{“,}\{\alpha_n\}\texttt{:}\{ar(\alpha_n)\}\text{”};
\end{array}
$$

- *A translation* $\mathsf{Compile}_t$ *from a term into a code string is defined, using a name table* $\mathrm{N}$ *which is defined in* $\mathsf{Compile}_c$*, as follows:*

$$
\begin{array}{lcl}
\mathsf{Compile}_t(x) & \stackrel{\text{def}}{=} & (\text{“”}, \mathrm{N}[x]) \\
\mid \mathsf{Compile}_t(\alpha(t_1, \ldots, t_n)) & \stackrel{\text{def}}{=} & \mathsf{let} \\
 & & \quad a = \mathsf{freshStr}(); \\
 & & \quad c = \text{“}\{a\}\texttt{=mkAgent(}\{\alpha\}\texttt{)}\text{”}; \\
 & & \quad (c_1,\, a_1) = \mathsf{Compile}_t(t_1); \\
 & & \quad c_1 = c_1 + \text{“}\{a\}\texttt{[1]=}\{a_1\}\text{”}; \\
 & & \quad \vdots \\
 & & \quad (c_n,\, a_n) = \mathsf{Compile}_t(t_n); \\
 & & \quad c_n = c_n + \text{“}\{a\}\texttt{[}n\texttt{]=}\{a_n\}\text{”}; \\
 & & \mathsf{in} \\
 & & \quad (c + c_1 + \cdots + c_n,\, a) \\
 & & \mathsf{end} \\
\mid \mathsf{Compile}_t(\$t) & \stackrel{\text{def}}{=} & \mathsf{let} \\
 & & \quad a = \mathsf{freshStr}(); \\
 & & \quad (c_1, a_1) = \mathsf{Compile}_t(t); \\
 & & \mathsf{in} \\
 & & \quad (c_1 \\
 & & \quad + \text{“}\{a\}\texttt{=mkInd()}\text{”} \\
 & & \quad + \text{“}\{a\}\texttt{[1]=}\{a_1\}\text{”}, a) \\
 & & \mathsf{end};
\end{array}
$$

- *A translation* $\mathsf{Compile}_i$ *from an interface* $\vec{u}$ *into a code sequence is defined as follows:*

$$
\begin{array}{lcl}
\mathsf{Compile}_i(\vec{u}) & \stackrel{\text{def}}{=} & \mathsf{let} \\
 & & \quad (c, n) = \mathsf{Compile}'_i(\vec{u}); \\
 & & \mathsf{in} \\
 & & \quad \text{“}\texttt{I=mkInterface[}\{n\}\texttt{]}\text{”} + c \\
 & & \mathsf{end};
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{Compile}'_i(-) & \stackrel{\text{def}}{=} & (\text{“”}, 0) \\
\mid \mathsf{Compile}'_i(u_1, \ldots, u_n) & \stackrel{\text{def}}{=} & \mathsf{let} \\
 & & \quad (c_1, a_1) = \mathsf{Compile}_t(u_1); \\
 & & \quad c_1 = c_1 + \text{“}\texttt{I[1]=}\{a_1\}\text{”}; \\
 & & \quad \vdots \\
 & & \quad (c_n, a_n) = \mathsf{Compile}_t(u_n); \\
 & & \quad c_n = c_n + \text{“}\texttt{I[}\{n\}\texttt{]=}\{a_n\}\text{”}; \\
 & & \mathsf{in} \\
 & & \quad (c_1 + \cdots + c_n, n) \\
 & & \mathsf{end};
\end{array}
$$

- *A translation* $\mathsf{Compile}_e$ *from an equation into a code string is defined as follows:*

$$
\begin{array}{lcl}
\mathsf{Compile}_e(t = s) & \stackrel{\text{def}}{=} & \mathsf{let} \\
 & & \quad (c_1, a_1) = \mathsf{Compile}_t(t); \\
 & & \quad (c_2, a_2) = \mathsf{Compile}_t(s); \\
 & & \quad c_3 = \text{“}\texttt{push(}\{a_1\}\texttt{,}\{a_2\}\texttt{)}\text{”}; \\
 & & \mathsf{in} \\
 & & \quad c_1 + c_2 + c_3 \\
 & & \mathsf{end};
\end{array}
$$

- *A translation* $\mathsf{Compile}_{es}$ *from an equation sequence into a code string is defined as follows:*

$$
\mathsf{Compile}_{es}(e_1, \ldots, e_n) \stackrel{\text{def}}{=} \mathsf{Compile}_e(e_1) + \cdots + \mathsf{Compile}_e(e_n);
$$

- *We define a translation* $\mathsf{Compile}_c$ *from a configuration* $\langle\ \vec{u} \mid \Delta\ \rangle$ *with a symbol set* $\Sigma$ *into a code string* $c$*, making a name table* $\mathrm{N}$ *which is used in* $\mathsf{Compile}_t$ *(called by* $\mathsf{Compile}_i$*, and* $\mathsf{Compile}_e$ *via* $\mathsf{Compile}_{es}$*), as follows:*

$$
\begin{array}{lcl}
\mathsf{Compile}_c(\Sigma, \langle\ \vec{u} \mid \Delta\ \rangle) & \stackrel{\text{def}}{=} & \mathsf{let} \\
 & & \quad c_0 = \mathsf{Compile}_s(\Sigma); \\
 & & \quad (\mathrm{N}, c_1) = \mathsf{makeN}(\mathsf{Name}(\langle\ \vec{u} \mid \Delta\ \rangle)); \\
 & & \quad c_2 = \mathsf{Compile}_i(\vec{u}); \\
 & & \quad c_3 = \mathsf{Compile}_{es}(\Delta); \\
 & & \mathsf{in} \\
 & & \quad c_0 + c_1 + c_2 + c_3 \\
 & & \mathsf{end};
\end{array}
$$

- *We write just* Compile *when there is no ambiguity.*

**Example 5.2.3**

*Let us take a configuration* $\langle\ r\ \mid\ \texttt{Add}(\texttt{Z}, r) = \texttt{S}(\texttt{Z})\ \rangle$ *with a symbol set* $\{\texttt{Z}, \texttt{S}, \texttt{Add}\}$ *as an example.*

*First, by applying* $\mathsf{Compile}_c$ *to this configuration, we obtain the following:*

$$\begin{aligned}
&\mathsf{Compile}_c(\{\texttt{Z}, \texttt{S}, \texttt{Add}\},\ \langle\ r\ \mid\ \texttt{Add}(\texttt{Z}, r) = \texttt{S}(\texttt{Z})\ \rangle) \quad = \\
&\qquad \mathsf{let} \\
&\qquad\quad c_0 = \mathsf{Compile}_s(\{\texttt{Z}, \texttt{S}, \texttt{Add}\}); \\
&\qquad\quad (\mathrm{N}, c_1) = \mathsf{makeN}(\{r\}); \\
&\qquad\quad c_2 = \mathsf{Compile}_i(r); \\
&\qquad\quad c_3 = \mathsf{Compile}_{es}(\texttt{Add}(\texttt{Z}, r) = \texttt{S}(\texttt{Z})); \\
&\qquad \mathsf{in} \\
&\qquad\quad c_0 + c_1 + c_2 + c_3 \\
&\qquad \mathsf{end}
\end{aligned}$$

*Next, we look at the expressions in the* let *clause precisely. In the first one, by unfolding* $\mathsf{Compile}_s(\{\texttt{Z}, \texttt{S}, \texttt{Add}\})$*, the following is obtained:*

```
#agent Z:0, S:1, Add:2
```

*By unfolding the second one* $\mathsf{makeN}(\{r\})$;*, the following is obtained:*

$$(\{(r, \texttt{r})\},\ \text{“}\texttt{r=mkName()}\text{”})$$

*By unfolding the third one* $\mathsf{Compile}_i(r)$*, the following is obtained:*

```
I=mkInterface[1]
I[1]=r
```

*By unfolding the last one* $\mathsf{Compile}_{es}(\texttt{Add}(\texttt{Z}, r) = \texttt{S}(\texttt{Z}))$*, we obtain the following:*

$$\begin{aligned}
&\mathsf{let} \\
&\quad (c_1,\ a_1) = \mathsf{Compile}_t(\texttt{Add}(\texttt{Z}, r)); \\
&\quad (c_2,\ a_2) = \mathsf{Compile}_t(\texttt{S}(\texttt{Z})); \\
&\quad c_3 = \text{“}\texttt{push}(\{a_1\},\{a_2\})\text{”}; \\
&\mathsf{in} \\
&\quad c_1 + c_2 + c_3 \\
&\mathsf{end}
\end{aligned}$$

*Here, in this* let *clause, we also look at the first two expressions. The following is an unfolding result of* $\mathsf{Compile}_t(\mathtt{Add}(\mathtt{Z}, r))$ *provided that* `a1` *is assigned as a fresh string:*

let
  $c_0 =$ "`a1=mkAgent(Add)`";
  $(c_1,\ a_1) = \mathsf{Compile}_t(\mathtt{Z})$;
  $c_1 = c_1 +$ "`a1[1]=`$\{a_1\}$";
  $(c_2,\ a_2) = \mathsf{Compile}_t(r)$;
  $c_2 = c_2 +$ "`a1[2]=`$\{a_2\}$";
in
  $(c_0 + c_1 + c_2,\ a_0)$
end

*Taking account of the following:*

- $\mathsf{Compile}_t(\mathtt{Z}) =$ ( "`a2=mkAgent(Z)`", `a2`),
- $\mathsf{Compile}_t(r) =$ ( "", `r`),

*the result of* $\mathsf{Compile}_t(\mathtt{Add}(\mathtt{Z}, r))$ *is obtained as follows:*

$(c_1,\ \mathtt{a1})$ *where* $c_1$ *is as follows:*

```
a1=mkAgent(Add)
a2=mkAgent(Z)
a1[1]=a2
a1[2]=r
```

*Regarding the next expression* $\mathsf{Compile}_t(\mathtt{S}(\mathtt{Z}))$*, we obtain the following unfolding result provided that* `b1` *is assigned as a fresh string:*

let
  $c_0 =$ "`b1=mkAgent(S)`";
  $(c_1,\ a_1) = \mathsf{Compile}_t(\mathtt{Z})$;
  $c_1 = c_1 +$ "`b1[1]=`$\{a_1\}$";
in
  $(c_0 + c_1,\ \mathtt{b1})$
end

*Here, taking account of the following:*

- $\mathsf{Compile}_t(\mathtt{Z}) = (\text{“}\mathtt{b2{=}mkAgent(Z)}\text{”}, \mathtt{b2})$

  *where* c2 *is assigned as a fresh string,*

*the result of* $\mathsf{Compile}_t(\mathtt{S(Z)})$ *is obtained as follows:*

$(c_2, \mathtt{b1})$ *where* $c_2$ *is as follows:*

```
b1=mkAgent(S)
b2=mkAgent(Z)
b1[1]=b2
```

*Thus, the result of* $\mathsf{Compile}_{es}(\mathtt{Add(Z}, r) = \mathtt{S(Z)})$ *is as follows:*

```
a1=mkAgent(Add)
a2=mkAgent(Z)
a1[1]=a2
a1[2]=r
b1=mkAgent(S)
b2=mkAgent(Z)
b1[1]=b2
push(a1,b1)
```

*Finally, collecting these results, we obtain the following sequence of code strings as a result of* $\mathsf{Compile}_c(\{\mathtt{Z}, \mathtt{S}, \mathtt{Add}\}, \langle\ r \mid \mathtt{Add(Z}, r) = \mathtt{S(Z)}\ \rangle)$*:*

```
#agent Z:0, S:1, Add:2
r=mkName()
I=mkInterface[1]
I[1]=r
a1=mkAgent(Add)
a2=mkAgent(Z)
a1[1]=a2
a1[2]=r
b1=mkAgent(S)
b2=mkAgent(Z)
b1[1]=b2
push(a1,b1)
```

### 5.2.2 Translation of interaction rules

Next, we define a compilation for rules.

**Definition 5.2.4 (Compilation of rules)**

*We define a translation* $\mathsf{Compile}_r$ *from a rule into a sequence of code strings, making a name table* N *which is used in* $\mathsf{Compile}_t$ *(called by* $\mathsf{Compile}_e$ *via* $\mathsf{Compile}_{es}$*), as follows:*

$$
\begin{array}{lll}
\mathsf{Compile}_r(\alpha(\vec{x}) = \beta(\vec{y}) \Rightarrow \Theta) & \stackrel{\text{def}}{=} & \mathsf{let} \\
& & \quad \mathrm{N}_1 = \mathsf{Compile}_{rn}(\vec{x}, \mathtt{L}, \emptyset); \\
& & \quad \mathrm{N}_2 = \mathsf{Compile}_{rn}(\vec{y}, \mathtt{R}, \mathrm{N}_1); \\
& & \quad (\mathrm{N}, c_1) = \mathsf{makeN'}(\mathsf{Name}(\Theta) - \{\vec{x}, \vec{y}\}, \mathrm{N}_2); \\
& & \quad c_2 = \mathsf{Compile}_{es}(\Theta); \\
& & \mathsf{in} \\
& & \quad \text{“}\texttt{rule } \{\alpha\} \ \{\beta\} \ \{\text{”} \\
& & \quad + \text{“}\texttt{stackFree()}\text{”} \\
& & \quad + c_1 + c_2 \\
& & \quad + \text{“}\texttt{free(L)}\text{”} \\
& & \quad + \text{“}\texttt{free(R)}\text{”} \\
& & \quad + \text{“}\}\text{”} \\
& & \mathsf{end};
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{Compile}_{rn}((x_1, \ldots, x_n),\ pos,\ \mathrm{N}) & \stackrel{\text{def}}{=} & \mathsf{let} \\
& & \quad \mathrm{N}_0 = \mathrm{N}; \\
& & \quad \mathrm{N}_1 = (\mathrm{N}_0[x_1] := \text{“}\{pos\}\texttt{[1]}\text{”}); \\
& & \quad \vdots \\
& & \quad \mathrm{N}_n = (\mathrm{N}_{n-1}[x_n] := \text{“}\{pos\}\texttt{[}\{n\}\texttt{]}\text{”}); \\
& & \mathsf{in} \\
& & \quad \mathrm{N}_n \\
& & \mathsf{end};
\end{array}
$$

**Example 5.2.5**

*Here, let us consider compilation of the following two rules:*

- $\mathtt{Add}(x_1, x_2) = \mathtt{Z} \Rightarrow x_1 = x_2$,
- $\mathtt{Add}(x_1, x_2) = \mathtt{S}(y) \Rightarrow x_2 = \mathtt{S}(w), \mathtt{Add}(x_1, w) = y$.

*First, we deal with the first rule* $\texttt{Add}(x_1, x_2) = \texttt{Z} \Rightarrow x_1 = x_2$. *By applying* $\mathsf{Compile}_r$ *into the rule, the following is obtained:*

$$
\begin{array}{l}
\mathsf{let} \\
\quad \mathrm{N}_1 = \mathsf{Compile}_{rn}((x_1, x_2), \texttt{L}, \emptyset); \\
\quad \mathrm{N}_2 = \mathsf{Compile}_{rn}((-), \texttt{R}, \mathrm{N}_1); \\
\quad (\mathrm{N}, c_1) = \mathsf{makeN'}(\emptyset, \mathrm{N}_2); \\
\quad c_2 = \mathsf{Compile}_{es}(x_1 = x_2)); \\
\mathsf{in} \\
\quad \text{“}\texttt{rule Add Z \{}\text{”} \\
\quad + \text{“}\texttt{stackFree()}\text{”} \\
\quad + c \\
\quad + \text{“}\texttt{free(L)}\text{”} \\
\quad + \text{“}\texttt{free(R)}\text{”} \\
\quad + \text{“}\texttt{\}}\text{”} \\
\mathsf{end}
\end{array}
$$

*Here, in this* let *clause, we look at the first three expressions. The following is a result after processing those:*

$$(\{(x_1, \texttt{L[1]}), (x_2, \texttt{L[2]})\},\ \text{“”})$$

*The following is an unfolding result of the last expression* $\mathsf{Compile}_{es}(x_1 = x_2)$*:*

$$
\begin{array}{l}
\mathsf{let} \\
\quad (c_1,\ a_1) = \mathsf{Compile}_t(x_1); \\
\quad (c_2,\ a_2) = \mathsf{Compile}_t(x_2); \\
\quad c_3 = \text{“}\texttt{push(\{}a_1\texttt{\},\{}a_2\texttt{\})}\text{”}; \\
\mathsf{in} \\
\quad c_1 + c_2 + c_3 \\
\mathsf{end}
\end{array}
$$

*Taking account of the following:*

- $\mathsf{Compile}_t(x_1) = (\text{“”},\ \texttt{L[1]})$
- $\mathsf{Compile}_t(x_2) = (\text{“”},\ \texttt{L[2]})$

*the result of* $\mathsf{Compile}_{es}(x_1 = x_2)$ *is obtained as follows:*

```
push(L[1],L[2])
```

*Thus, the result of* $\mathsf{Compile}_r(\mathtt{Add}(x_1, x_2) = \mathtt{Z} \Rightarrow x_1 = x_2)$ *is obtained as follows:*

```
rule Add Z {
  stackFree()
  push(L[1],L[2])
  free(L)
  free(R)
}
```

*Secondly, we deal with the other rule* $\mathtt{Add}(x_1, x_2) = \mathtt{S}(y) \Rightarrow x_2 = \mathtt{S}(w), \mathtt{Add}(x_1, w) = y$. *By applying* $\mathsf{Compile}_r$ *into the rule, the following is obtained:*

$$
\begin{array}{l}
\mathsf{let} \\
\quad \mathrm{N}_1 = \mathsf{Compile}_{rn}((x_1, x_2), \mathtt{L}, \emptyset); \\
\quad \mathrm{N}_2 = \mathsf{Compile}_{rn}((y), \mathtt{R}, \mathrm{N}_1); \\
\quad (\mathrm{N}, c_1) = \mathsf{makeN}'(\{w\}, \mathrm{N}_2); \\
\quad c_2 = \mathsf{Compile}_{es}(x_2 = \mathtt{S}(w), \mathtt{Add}(x_1, w) = y); \\
\mathsf{in} \\
\quad \text{“}\texttt{rule Add S \{}\text{”} \\
\quad + \text{“}\texttt{stackFree()}\text{”} \\
\quad + c \\
\quad + \text{“}\texttt{free(L)}\text{”} \\
\quad + \text{“}\texttt{free(R)}\text{”} \\
\quad + \text{“}\}\text{”} \\
\mathsf{end}
\end{array}
$$

*Here, we look the first three expressions in this* let *clause. The following is a result after processing those:*

$$(\{(x_1, \mathtt{L[1]}), (x_2, \mathtt{L[2]}), (y, \mathtt{R[1]}), (w, \mathtt{w})\}, \text{“}\texttt{w=mkName()}\text{”})$$

*By unfolding the last expression* $\mathsf{Compile}_{es}(x_2 = \mathtt{S}(w), \mathtt{Add}(x_1, w) = y)$ *in the* let *clause,*

*we obtain the following:*

$$
\begin{array}{l}
\mathsf{let} \\
\quad c_1 = \mathsf{Compile}_e(x_2 = \mathtt{S}(w)); \\
\quad c_2 = \mathsf{Compile}_e(\mathtt{Add}(x_1, w) = y); \\
\mathsf{in} \\
\quad c_1 + c_2 \\
\mathsf{end}
\end{array}
$$

*In this* let *clause, we examine the first expression* $\mathsf{Compile}_e(x_2 = \mathtt{S}(w))$. *The following is obtained by unfolding it:*

$$
\begin{array}{l}
\mathsf{let} \\
\quad (c_1,\, a_1) = \mathsf{Compile}_t(x_2); \\
\quad (c_2,\, a_2) = \mathsf{Compile}_t(\mathtt{S}(w)); \\
\quad c_3 = \text{“}\mathtt{push(\{}a_1\mathtt{\},\{}a_2\mathtt{\})}\text{”}; \\
\mathsf{in} \\
\quad c_1 + c_2 + c_3 \\
\mathsf{end}
\end{array}
$$

*Taking account of the following:*

- $\mathsf{Compile}_t(x_2) = (\text{“”},\, \mathtt{L[2]})$
- $\mathsf{Compile}_t(\mathtt{S}(w)) = (c_1,\, \mathtt{aS})$ *where* $c_1$ *is as follows:*

```
aS=mkAgent(S)
aS[1]=w
push(L[2],aS)
```

*the result of* $\mathsf{Compile}_e(x_2 = \mathtt{S}(w))$ *is obtained as follows:*

```
aS=mkAgent(S)
aS[1]=w
push(L[2],aS)
```

*Regarding the second expression* $\mathsf{Compile}_e(\mathtt{Add}(x_1, w) = y)$, *we obtain the following by*

*unfolding it:*

$$
\begin{array}{l}
\mathsf{let} \\
\quad (c_1,\, a_1) = \mathsf{Compile}_t(\mathtt{Add}(x_1, w)); \\
\quad (c_2,\, a_2) = \mathsf{Compile}_t(y); \\
\quad c_3 = \text{“}\mathtt{push}(\{a_1\},\{a_2\})\text{”}; \\
\mathsf{in} \\
\quad c_1 + c_2 + c_3 \\
\mathsf{end}
\end{array}
$$

*The same as the above, we can obtain the following as a result:*

```
aAdd=mkAgent(Add)
aAdd[1]=L[1]
aAdd[2]=w
push(aAdd,R[1])
```

*Thus, we obtain a code sequence as a result of* $\mathsf{Compile}_r(\mathtt{Add}(x_1, x_2) = \mathtt{S}(y) \Rightarrow x_2 = \mathtt{S}(w), \mathtt{Add}(x_1, w) = y)$ *as follows:*

```
rule Add S {
  stackFree()
  w=mkName()
  aS=mkAgent(S)
  aS[1]=w
  push(L[2],aS)
  aAdd=mkAgent(Add)
  aAdd[1]=L[1]
  aAdd[2]=w
  push(aAdd,R[1])
  free(L)
  free(R)
}
```

## 5.3 Execution model in the C language

In this section, we explain how these translated codes in Section 5.2.2 are evaluated on the standardised implementation model in the C language, showing correspondence of codes in LL0 with ones in the C language.

### 5.3.1 Implementation of instructions

In this section we explain how each instruction in Figure 5.3 corresponds to the C language codes in the standardised implementation model.

- `#agent` $\alpha_1 : p_1, \ldots, \alpha_n : p_n$

  For each sort of agent, we assign a unique number that is greater than 1. The declaration for agent symbols corresponds as follows:

  ```
  #define ID_α1 1
   ⋮
  #define ID_αn n
  #define MAX_AGENTID n
  ```

  In addition, to manage symbol characters and arities, we define two arrays named `Symbols` and `Arities` respectively as follows:

  ```
  char Symbols[MAX_AGENTID+1] = {"", "α1",...,"αn"};
  int Arities[MAX_AGENTID+1] = {1,p1,...,pn};
  ```

- `I=mkInterface(`$n$`)`

  This makes a global $n$-size array for the interface, and corresponds to the following codes:

  ```
  #define SIZE_INTERFACE n
  Agent *I[SIZE_INTERFACE];
  ```

- $x$ `=mkAgent(`$id$`)`

  This makes a variable $x$ whose type is `Agent` pointer and assigns an agent node whose `id` is $id$. This corresponds to the following codes:

  ```
  Agent *x=mkAgent(id);
  ```

- $x$ `=mkName()`

  This makes a variable $x$ whose type is `Agent` pointer and assigns a name node. This corresponds to the following codes:

  ```
  Agent *x=mkName();
  ```

- $x$ = `mkInd()`

  This makes a variable $x$ whose type is `Agent` pointer and assigns an indirection node.

  ```
  Agent *x=mkAgent(ID_INDIRECTION);
  ```

- `free(`$x$`)`

  This disposes of a graph node assigned to $x$. This corresponds to the following code:

  ```
  freeAgent(x);
  ```

- $x$`[`$p$`]=`$y$

  This assigns a graph element $y$ to a port $p$ of an agent node $x$. The port $p$ in LL0 corresponds to the port $p-1$ in the standardised implementation method, and thus this instruction corresponds to the following code:

  ```
  x[p-1]=y;
  ```

- $x$`[0]=`$\alpha$

  This changes the id of an agent $x$ into $\alpha$. This corresponds to the following codes:

  ```
  x->id=ID_α;
  ```

- `push(`$x$`,`$y$`)`

  This pushes two agents onto the equation stack. This corresponds to the following codes:

  ```
  pushActive(x,y);
  ```

- `stackFree()`

  This disposes of the top element of the equation stack. In the translation result, it occurs in rule procedures. In the standardised implementation method, the function `popActive` manages the index of the equation stack, and thus no code is required.

### 5.3.2 Implementation of rule procedures

Next, we manage the translated LL0 instructions for rule procedures.

A rule procedure in LL0 such as

```
rule Alpha Beta
```

is encoded as a function that is named as `Alpha_Beta`, takes two pointers `*a1` and `*a2` to two elements of the equation, and creates nets according to interaction rules. The special variables `L` and `R` in the rule procedures are denoted as `*a1` and `*a2`, and thus

$$\texttt{L[1]}, \texttt{L[2]}, \ldots, \texttt{R[1]}, \texttt{R[2]}, \ldots$$

are expressed as

$$\texttt{a1->port[0]}, \texttt{a1->port[1]}, \ldots, \texttt{a2->port[0]}, \texttt{a2->port[1]}, \ldots$$

.

For instance, the rule procedure for `Add` and `Z` in Section 5.1.2 is encoded as follows:

```
void Add_Z(Agent *a1, Agent *a2) {
  pushActive(a1->port[0], a1->port[1]);
  freeAgent(a1);
  freeAgent(a2);
}
```

The rule procedure for `Add` and `S` is encoded as follows:

```
void Add_S(Agent *a1, Agent *a2) {
  Agent *aS = mkAgent(ID_S);
  Agent *aAdd = mkAgent(ID_Add);
  Agent *w = mkName();
  aAdd->port[0]=a1->port[0];
  aAdd->port[1]=w;
  pushActive(aAdd, a2->port[0]);
  aS->port[0]=w;
  pushActive(a1->port[1], aS);
  freeAgent(a1);
  freeAgent(a2);
}
```

To manage these functions, we define a rule table `R`, which stores pointers to those functions. Here, for simplicity, we use the following simple matrix:

```
typedef void (*RuleFun)(Agent *a1, Agent *a2);
RuleFun R[MAX_AGENTID+1][MAX_AGENTID+1];
```

For instance, the above functions are stored as follows:

```
R[ID_Add][ID_Z] = &Add_Z;
R[ID_Add][ID_S] = &Add_S;
```

## 5.4 Execution model in a bytecode interpreter

In this section, as another correspondence of instruction sets in LL0, we introduce a bytecode interpreter that can evaluate the translated LL0 code in Section 5.2.

This interpreter is built on the standardised implementation method, and thus it has the same data-structure in the C language. These bytecodes are regarded as intermediate codes of an interpreter of interaction nets, and those are evaluated by a register-based virtual machine such as the Lua's virtual machine [33]. The bytecodes mainly control the following components:

- Registers `Reg`,
- Global array `G` for the interface,
- Equation stack.

We use the following notations: `Reg`($n$), `G`($m$) mean the $n$th Register and the $m$th element of Global array respectively. We do not care about the sizes of the `R` and `G`.

The `Reg` and `G` are implemented by `Agent*` arrays. Each Register `Reg`($n$) is used for each variable in LL0 and the Global array `G` is used as the interface array.

The Figure 5.5 illustrates the bytecodes of the interpreter. We write a sequence of bytecodes with the space and breakline separators such as:

```
MKAGENT 11 3
MKAGENT 12 4
PUSH 11 12
```

The code `RETURN` is used to return the execution call for these bytecodes by a runtime function.

These codes are defined as the following integer constants in the standardised implementation method:

```
enum {MKAGENT,MKNAME,MKIND,FREE,MOVEP,MOVEG,CHGID,PUSH,RETURN};
```

### 5.4.1 Implementation of instructions

We explain how each instruction in LL0 (shown in Figure 5.3), which is required for the translation from the nets, corresponds to the bytecodes below:

| Bytecode | Description |
|---|---|
| `MKAGENT A B` | `Reg(A):=mkAgent(B)` |
| `MKNAME A` | `Reg(A):=mkName()` |
| `MKIND A` | `Reg(A):=mkInd()` |
| `FREE A` | `freeAgent(Reg(A))` |
| `MOVEP A B C` | `Reg(A)->port[B]:=Reg(C)` |
| `MOVEG A B` | `G(A):=Reg(B)` |
| `CHGID A B` | `Reg(A)->id:=B` |
| `PUSH A B` | `pushActive(Reg(A),Reg(B))` |
| `RETURN` | Return from the execution call. |

Figure 5.5: Instructions of a bytecode interpreter

- `#agent` $\alpha_1 : p_1, \ldots, \alpha_n : p_n$

  This model used the standardised implementation model, so it is the same as the correspondence in the C language codes as follows;

  `#define ID_`$\alpha_1$ `1`
  $\vdots$
  `#define ID_`$\alpha_n$ $n$
  `#define MAX_AGENTID` $n$
  `char Symbols[MAX_AGENTID+1] = {"", "`$\alpha_1$`",...,"`$\alpha_n$`"};`
  `int Arities[MAX_AGENTID+1] = {1,`$p_1, \ldots, p_n$`};`

- `I=mkInterface(`$n$`)`

  This makes a global $n$-size array for the interface. The interface is managed by the Global array `G`, and thus there is no corresponding code.

- $x$ `=mkAgent(`$id$`)`

  We assume that each variable $x$ is assigned to a Register `Reg(`$n$`)`, and this instruction corresponds to the following codes:

  `MKAGENT` $n$ $id$

- $x$ `=mkName()`

  The same as the case of `mkAgent`, this instruction corresponds to the following codes:

  `MKNAME` $n$

- $x$ = `mkInd()`

  This is the same as the case of `mkName`:

  `MKIND` $n$

- `free(`$x$`)`

  This instruction directly corresponds to the following code, where $x$ is assigned to a Register `Reg(`$n$`)`:

  `FREE` $n$

- $x$`[`$p$`]=`$y$

  When $x$ and $y$ are assigned to Registers `Reg(`$m$`)` and `Reg(`$n$`)`, it corresponds to the following codes, where $q$ is a constant such as $q = p - 1$:

  `MOVEP` $m$ $q$ $n$

- $x$`[0]=`$\alpha$

  When $x$ are assigned to Registers `Reg(`$n$`)`, this corresponds to the following codes:

  `CHGID` $n$ `ID_`$\alpha$

- `I[`$p$`]=`$y$

  When $y$ are assigned to Registers `Reg(`$m$`)`, it corresponds to the following code:

  `MOVEG` $p$ $m$

- `push(`$x$`,`$y$`)`

  This instruction directly corresponds to the following code, when $x$ and $y$ are assigned to Registers `Reg(`$m$`)` and `Reg(`$n$`)`:

  `PUSH` $m$ $n$

- `stackFree()`

  The same as the correspondence to the standardised implementation model, this is managed by the function `popActive`, and thus no corresponding code is required:

A sequence of bytecodes is evaluated by the function `evalCode` as follows:

```
void evalCode(int *code) {
  int pc=0; // program counter
  while (1) {
    switch(code[pc]) {
      case MKAGENT:
        Reg[code[pc+1]]=mkAgent(code[pc+2]);
        pc+=3;
        break;
      case MKNAME:
        Reg[code[pc+1]]=mkName();
        pc+=2;
        break;
      case MKIND:
        Reg[code[pc+1]]=mkInd();
        pc+=2;
        break;
      case FREE:
        freeAgent(Reg[code[pc+1]]);
        pc+=2;
        break;
      case MOVEP:
        Reg[code[pc+1]]=(Reg[code[pc+2]])->port[Reg[code[pc+3]]];
        pc+=4;
        break;
      case MOVEG:
        G[code[pc+1]]=Reg[code[pc+2]];
        pc+=3;
        break;
      case CHGID:
        (Reg[code[pc+1]])->id=code[pc+2];
        pc+=3;
        break;
      case PUSH:
        pushActive(Reg[code[pc+1]],Reg[code[pc+2]]);
```

```
            pc+=3;
            break;
          case RETURN:
            return;
            break;
        }
      }
    }
```

### 5.4.2 Implementation of rule procedures

Next, we explain how the translated rule procedures are realised in the bytecode interpreter.

A rule procedure has a sequence of instructions that contains the following special variables

$$\texttt{L[1]}, \texttt{L[2]}, \texttt{L[3]}, \ldots, \texttt{R[1]}, \texttt{R[2]}, \texttt{R[3]}, \ldots$$

to refer to the ports of the active pairs. We assign those variables into the Registers `Reg` as

$$0, 1, 2, \ldots, \texttt{MAX\_PORT}, \texttt{MAX\_PORT} + 1, \texttt{MAX\_PORT} + 2, \ldots$$

respectively, and starts the numbering for variables in the Registers from $\texttt{MAX\_PORT} \times 2$. For a rule procedure in LL0 such as

```
rule Alpha Beta
```

we store an integer array `Alpha_Beta` with a sequence of codes.

For instance, a sequence of instructions of the rule procedure for `Add` and `Z` in Section 5.1.2 is encoded as follows:

```
PUSH 0 1            // push(L[1],L[2])
RETURN
```

To make the correspondence clear, we add a corresponding instruction in LL0 as a comment. The disposing of the active pair is managed by the runtime function `eval`. These codes are executed by a runtime function, and we put `RETURN` at the end of codes to show the termination of the codes. Those are stored in an integer array `Add_Z` as follows:

```
int *Add_Z = {PUSH, 0, 1, RETURN};
```

The instruction sequence of the rule procedure for `Add` and `S`, when we assume that `MAX_PORT` is 5, is encoded as follows:

```
MKAGENT 10 ID_S     // aS=mkAgent(S)
MKAGENT 11 ID_Add   // aAdd=mkAgent(Add)
MKNAME 12           // w=mkName()
MOVEP 11 0 0        // aAdd[1]=L[1]
MOVEP 11 1 12       // aAdd[2]=w
PUSH 11 5           // push(aAdd, R[1])
MOVEP 10 0 12       // aS[1]=w
PUSH 1 10           // push(L[2], aS)
RETURN
```

and these are stored into an integer array `Add_S`:

```
int *Add_S = {MKAGENT, 10, ID_S, MKAGENT, 11, ID_Add, MKNAME, 12,
  MOVEP, 11, 0, 0, MOVEP, 11, 1, 12, PUSH, 11, 5, MOVEP, 10, 0, 12,
  PUSH, 1, 10, RETURN};
```

To manage these codes, we define a code table `Code`, which stores codes for rule procedures. Here, for simplicity, we use the following simple matrix:

```
int *Code[MAX_AGENTID+1][MAX_AGENTID+1];
```

The code table is initialised by the function `initRuleTable` as follows:

```
void initRuleTable() {
  int i,j;
  for (i=0; i<= MAX_AGENTID; i++)
    for (j=0; j<= MAX_AGENTID; j++)
      Code[i][j] = NULL;
  /* interaction rules */
  Code[ID_a][ID_b]=a_b;
    ⋮
}
```

Assignments of code sequences for interaction rules to the `Code` are declared below the comment line `/* interaction rules */`. For instance, the functions `Add_Z` and `Add_S` are written as follows:

```
  /* interaction rules */
  Code[ID_Add][ID_Z] = Add_Z;
  Code[ID_Add][ID_S] = Add_S;
```

Those code sequences are referred by the runtime function `eval` is written as follows:

```
void eval() {
  Agent *a1, *a2;
  while (popActive(&a1, &a2)) {
    if (a2->id != ID_NAME) {
      if (a1->id != ID_NAME) {
        int i;
        for (i=0; i<Arities[a1->id]; i++) //For L[0],L[1],L[2],...
          Reg[i]=a1->port[i];
        for (i=0; i<Arities[a2->id]; i++) //For R[0],R[1],R[2],...
          Reg[MAX_PORT+i]=a2->port[i];
        evalCode(Code[a1->id][a2->id]); //Evaluate code sequences
        freeAgent(a1); freeAgent(a2);
      } /* The below is operations for x=t */
          ⋮
    } else {
      /* The below is operations for t=y and x=y */
        ⋮
    }
  }
}
```

## 5.5 Summary

In this chapter we proposed the low-level language LL0. Instructions in LL0 not only have almost one-to-one correspondence to the standardised implementation model, but also are considered bytecodes of a virtual machine. We also introduced a compilation from interaction nets to LL0. This compilation and bytecode aspect leads to our new interpreter for interaction nets introduced in Chapter 7.

# Chapter 6

# A language for programming in interaction nets

Programming with pure interaction nets is analogous to programming in the pure $\lambda$-calculus. They lack datatypes and constructs that one would expect in a typical programming language. In this chapter we take a step towards extending interaction nets from their pure form and to allow them to facilitate nested pattern matching, built-in datatypes and operations over these types.

First we extend interaction rules so that nested pattern matching can be performed. This extension is conservative—the extended rules can be translated back into ordinary rules in pure interaction nets. Next we introduce agents that may optionally contain attributes, which are values of base type: integers, and interaction rules with these attributes and conditions. Finally, we extend the execution model to make use of these extensions.

## 6.1 Pattern matching

### 6.1.1 Motivations

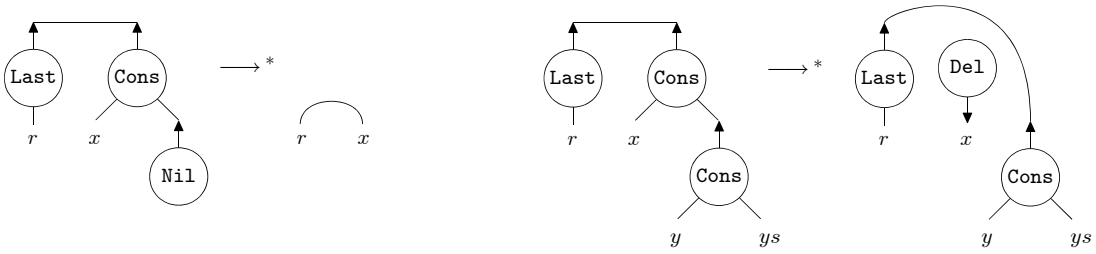
In this section, we motivate our work by investigating how we can translate a function with pattern matching into interaction nets.

If we consider a functional programming language as an orthogonal term rewriting system, we can translate programs into interaction nets [15]. In this way, if we take both the name of the function and the first argument as agents, we can represent these programs as computations in interaction nets. For example, the following function `Last` that returns the last element of a given list:

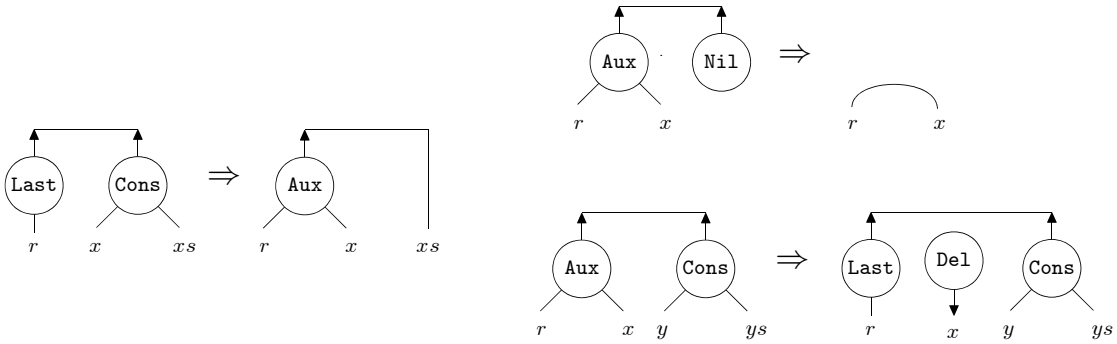
```
fun Last [x] = x
```

```
| Last (x::xs) = Last xs;
```

can be represented as the following computations:




These computations, however, are not defined by interaction rules directly because these LHS nets in the above graphs require other agents (`Nil` and `Cons`) that are connected to their active pairs. By introducing an auxiliary agent these can be realised as interaction rules:




This set of rules will compute and return the last element of a list. We argue that the introduction of the auxiliary agents to the system is not satisfactory from a programmer's perspective. Programmers want to write simpler programs rather than more complicated ones. To solve this problem, we extend the definition of rules to facilitate nested pattern matching.

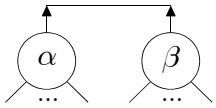
### 6.1.2 Interaction rules for nested patterns (INP)

In this section we present our framework INP that extends ordinary interaction rules (ORN) so that we can perform rewritings between nested agents. The main difference from ORN is that we allow the left hand side of a rule to contain more than two agents. The definitions of agents and nets remain the same as for ORN.

**Definition 6.1.1**

*A* nested active pair *$P$ is inductively defined as follows:*

**Base:** *Every active pair in ORN is a nested active pair. A nested active pair for an active pair $(\alpha, \beta)$:*

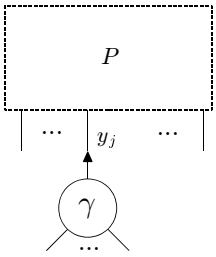



*is represented textually as follows:*

$$\langle \alpha(\vec{x}) \bowtie \beta(\vec{y}) \rangle$$

.

**Step:** *A net obtained as a result of connecting the principal port of some agent to a free port in a nested active pair $P$ is also a nested active pair. For instance, the following net such that an agent $\gamma$ is connected to a free port $y_j$ of a nested active pair $P$ is also a nested active pair:*




*We represent this textually as*

$$\langle P,\ y_j - \gamma(\vec{z}) \rangle\,.$$

**Definition 6.1.2**

*An interaction rule in INP is given by $P \Rightarrow N$ where $P$ is a nested active pair. All the free ports are preserved during reduction, and there is at most one rule with $P$ in any given system.*

**Proposition 6.1.3**

*Let $R$ be a rule in ORN, then $R$ is also a rule in INP.*

**Proof.** All rules $P \Rightarrow N$ where $P$ contains just two agents (active pair) are valid ORN rules. These active pairs fall into the base definition of nested active pairs. □

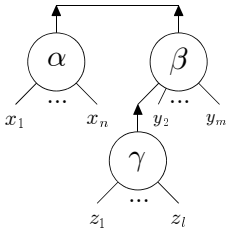
We aim to extend ORN in a conservative way and retain the property of strong confluence. For this purpose, we introduce a condition that restricts the formation of the set of interaction rules in INP.

**Definition 6.1.4**

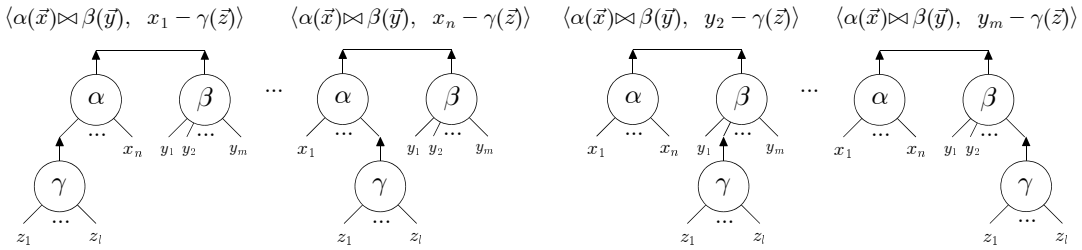
*A set of nested active pairs $\mathcal{P}$ is* sequential *if and only if, when $\langle P,\ y_j - \gamma(\vec{z}) \rangle \in \mathcal{P}$, then*

- *for the nested pair $P$, $P \in \mathcal{P}$ and,*
- *for all free ports $y$ in $P$ except the $y_j$ and for all agents $\alpha$, $\langle P,\ y - \alpha(\vec{w})\rangle \notin \mathcal{P}$.*

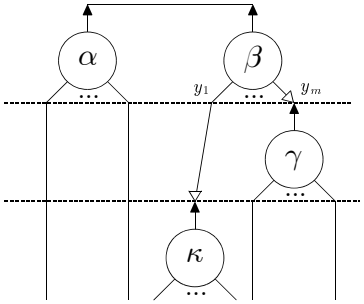
As an example, consider the following nested active pair $P$ in a sequential set $\mathcal{P}$:




represented textually as $\langle \alpha(x_1, \ldots, x_n) \bowtie \beta(y_1, \ldots, y_m),\ y_1 - \gamma(z_1, \ldots, z_l)\rangle$. Then we can not have any other nested active pair $(\alpha, \beta)$ such that the port $y_1$ is free. Thus, the following definitions violate the condition of the set $\mathcal{P}$:




For clarity, we draw lines and triangles on auxiliary ports that connect to nested agents. As an example, we represent a nested active pair $\langle \alpha(\vec{x}) \bowtie \beta(\vec{y}),\ y_m - \gamma(\vec{z}),\ y_1 - \kappa(\vec{w})\rangle$ graphically as follows:




Note that this nested active pair belongs to the set $\mathcal{P}$ because $P \in \mathcal{P}$.

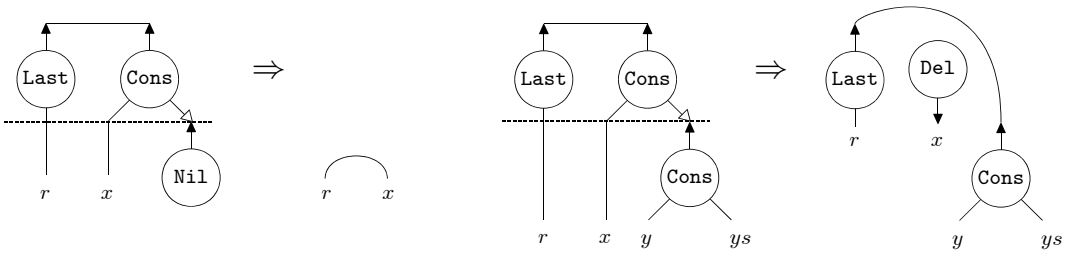
**Definition 6.1.5**

*A set of rules $\mathcal{R}$ in INP is* well-formed *if and only if,*

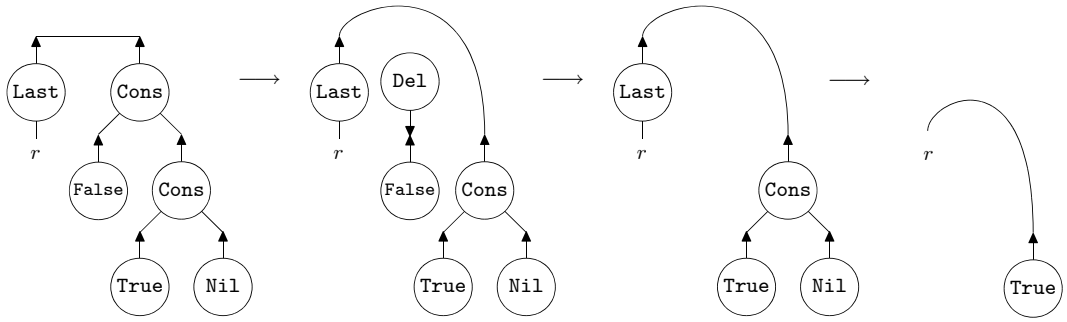
- *there is a sequential set which contains every nested active pair of the LHS in $\mathcal{R}$,*
- *for every rule $P \Rightarrow N$ in $\mathcal{R}$, there is no interaction rule $P' \Rightarrow N'$ in $\mathcal{R}$ such that $P'$ is a subnet of $P$.*

**Example 6.1.6**

*The computation in Section 6.1.1 is defined as a well-formed rule set:*




*and the following computation can be performed:*



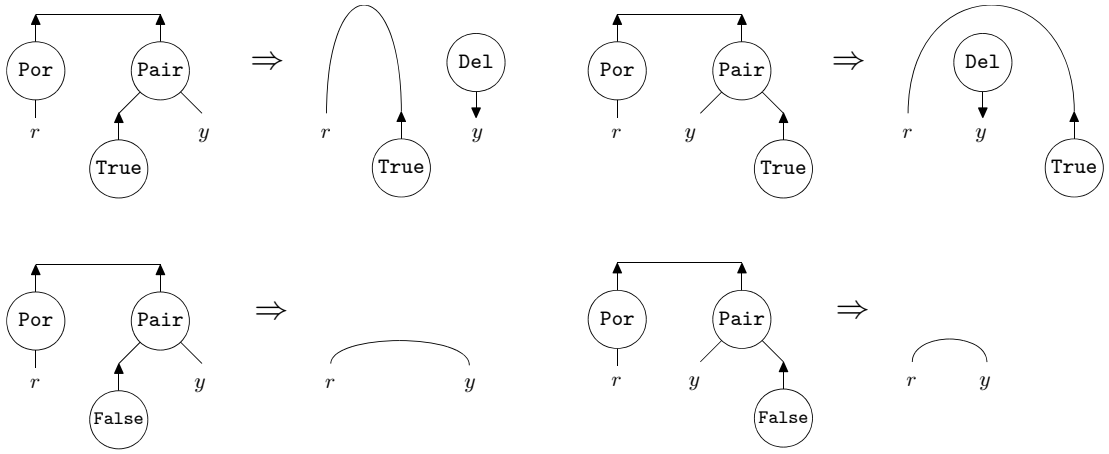

In the above example, the rewriting is strongly confluent because there is no *critical pair*. We loose this property if there are more than two rules that can be applied to the same net.

**Example 6.1.7**

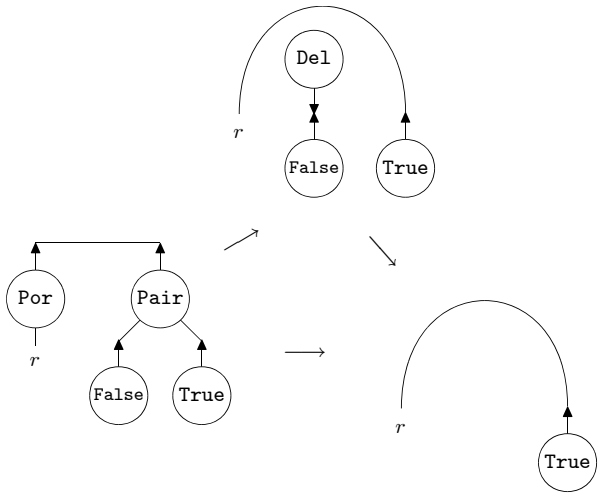
*We can encode the following definition of the parallel-or function* `Por`*:*

$$\begin{aligned}
\texttt{Por}(\texttt{True}, y) &= \texttt{True} \\
\texttt{Por}(y, \texttt{True}) &= \texttt{True} \\
\texttt{Por}(\texttt{False}, y) &= y \\
\texttt{Por}(y, \texttt{False}) &= y
\end{aligned}$$

*as a set of INP rules:*

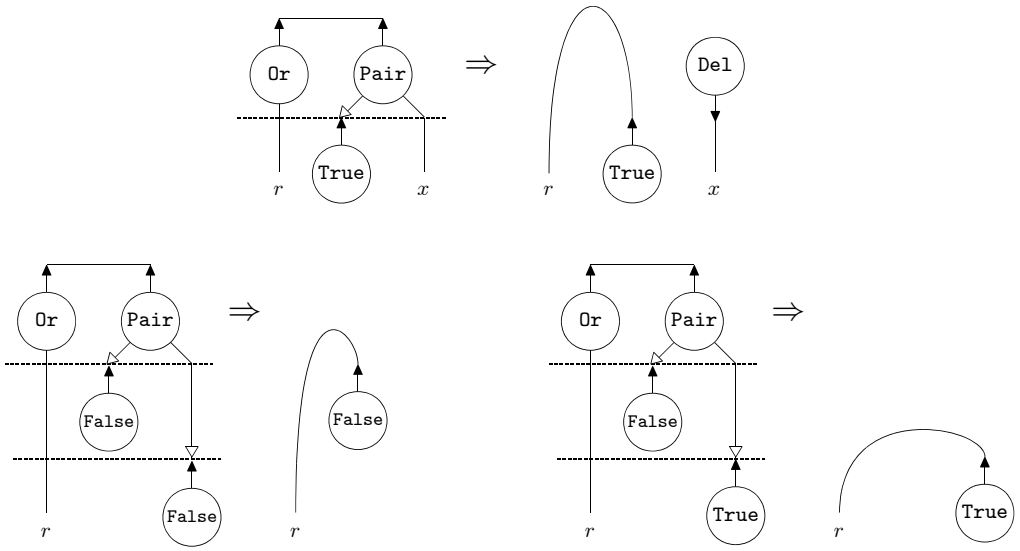


*However, this is not a well-formed set of rules because there is no sequential set which contains both* $\langle \texttt{Por}(x) \bowtie \texttt{Pair}(y_1, y_2),\ y_1 - \texttt{True} \rangle$ *and* $\langle \texttt{Por}(x) \bowtie \texttt{Pair}(y_1, y_2),\ y_2 - \texttt{True} \rangle$. *Therefore, the reduction is not strongly confluent (but still confluent in this example):*



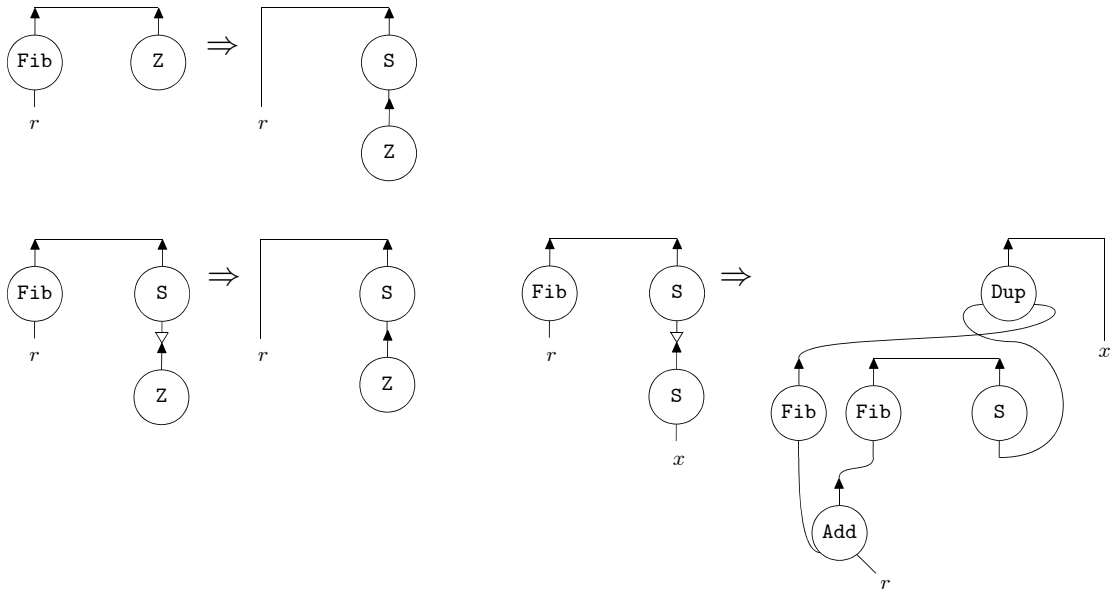

*On the other hand, the following rule set for the* `Or` *function is well-defined:*

$$\begin{aligned} \texttt{Or}(\texttt{True}, x) &= \texttt{True} \\ \texttt{Or}(\texttt{False}, \texttt{False}) &= \texttt{False} \\ \texttt{Or}(\texttt{False}, \texttt{True}) &= \texttt{True} \end{aligned}$$

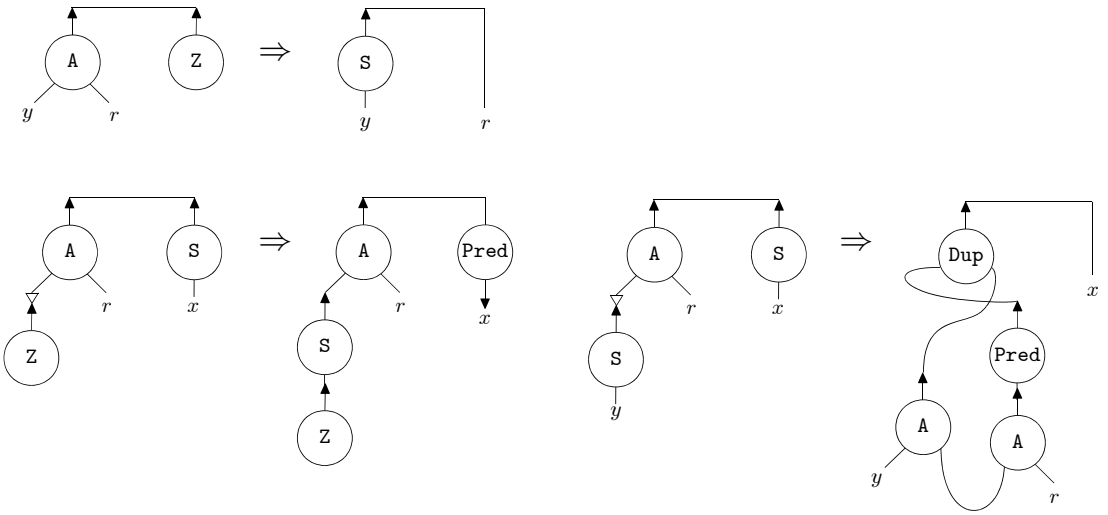



**Example 6.1.8**

*Interaction rules for Fibonacci number in Figure 2.4 are written with nested pattern matching as follows:*




*Interaction rules for Ackermann function in Figure 2.5 are written with nested pattern matching as follows:*

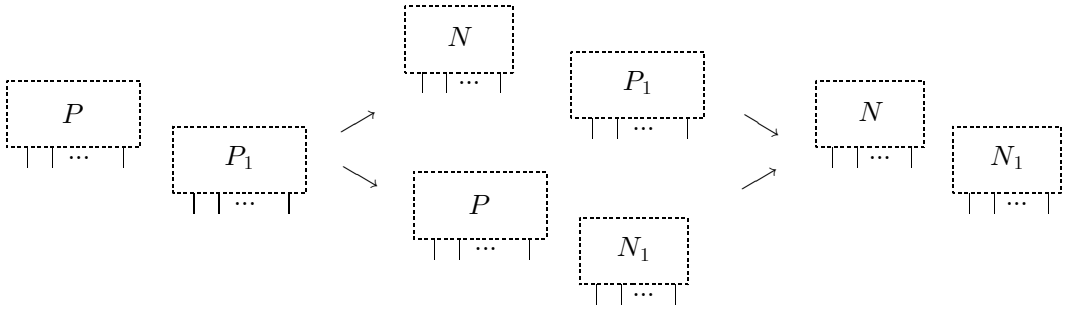



**Proposition 6.1.9 (Strong Confluence)**

*If a given rule set $\mathcal{R}$ in INP is well-formed, then the reduction in $\mathcal{R}$ is strongly confluent.*

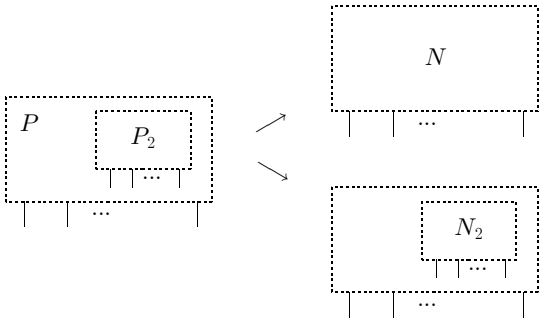
**Proof.** Assume that $P \Rightarrow N \in \mathcal{R}$. There are two cases where critical pairs can arise for a net which contains $P$:

**case 1:** there is no overlap between rules. We assume that there is a rule $P_1 \Rightarrow N_1 \in \mathcal{R}$ where $P_1$ does not overlap with $P$. In this case, the reduction is strongly confluent:



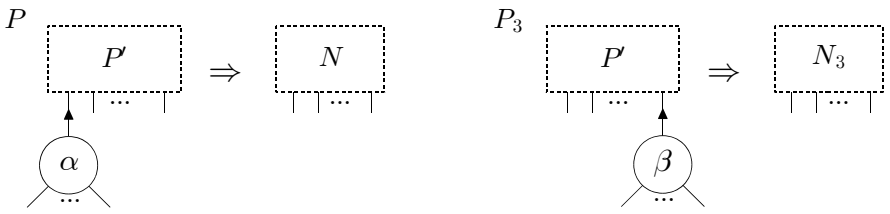

**case 2:** there are overlaps between rules.

**case 2.1:** We assume that there is a rule $P_2 \Rightarrow N_2 \in \mathcal{R}$ where $P_2$ is a subnet of $P$.




This case can not arise if $\mathcal{R}$ is well formed. Therefore $P_2 \Rightarrow N_2 \notin \mathcal{R}$

**case 2.2:** We assume that there is a rule $P_3 \Rightarrow N_3 \in \mathcal{R}$ where $P_3$ contains a subnet of $P$.

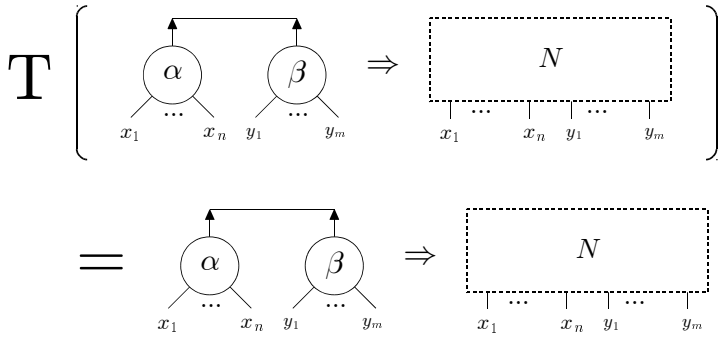



There is no sequential set which contains both $P$ and $P_3$, therefore $P_3 \Rightarrow N_3 \notin \mathcal{R}$. □

### 6.1.3 Translation

In this section, we define the translation function **T** from interaction rules with nested active pairs $P \Rightarrow N$ to interaction rules with only active pairs:

- When $P$ is just an active pair $\langle \alpha(x_1, \ldots, x_n) \bowtie \beta(y_1, \ldots, y_m) \rangle$, then the translation **T** is the identity:




- When $P$ is a nested active pair such that

$$P = \langle \alpha(p_1, \ldots, p_w) \bowtie \beta(q_1, \ldots, q_k, \ldots, q_u),\ q_k - \gamma(z_1, \ldots, z_l),\ \vec{a} \rangle$$

where $\vec{a}$ is a sequence of agents connections such that $r_i - \tau(\vec{w})$, then the translation **T** generates the following rules:

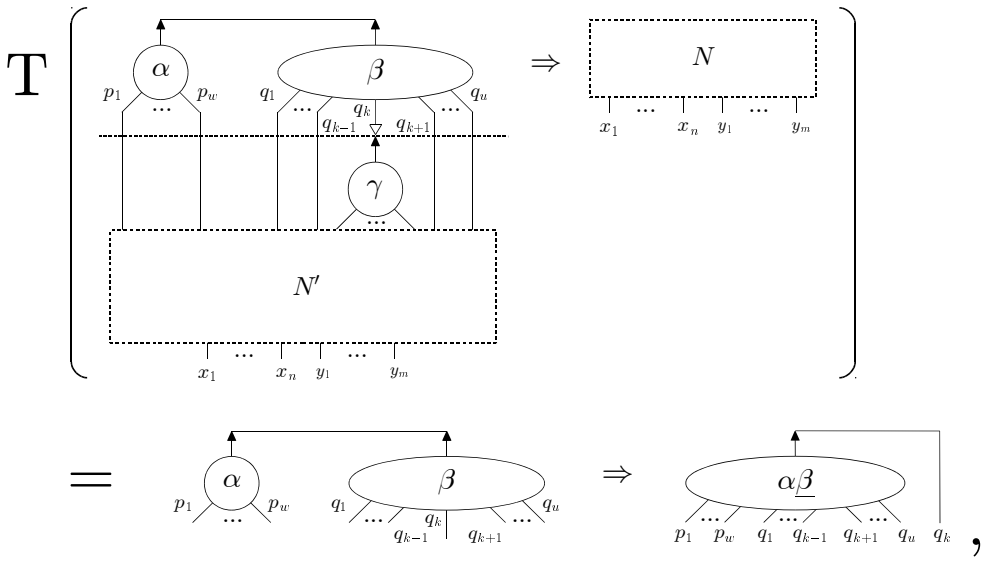
  - $\alpha(p_1, \ldots, p_w) = \beta(q_1, \ldots, q_k, \ldots, q_u)$

    $\Rightarrow q_k = \alpha\underline{\beta}(p_1, \ldots, p_w, q_1, \ldots, q_{k-1}, q_{k+1}, \ldots, q_u)$

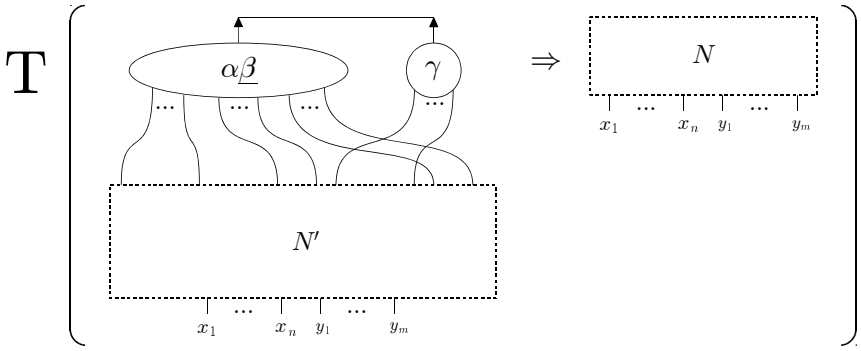
    where $\alpha\underline{\beta}$ is a new agent named from a concatenation of the LHS nested active pair agents. Since $q_k$ is connected to the principal port of $\gamma$, an active pair $(\alpha\underline{\beta}, \gamma)$ will be formed.

  - $\langle \alpha\underline{\beta}(p_1, \ldots, p_w, q_1, \ldots, q_{k-1}, q_{k+1}, \ldots, q_u) \bowtie \gamma(z_1, \ldots, z_l),\ \vec{a} \rangle \Rightarrow N$. This rule is recursively translated to obtain a rule with just an active pair.

    Graphically, this translation is given by:

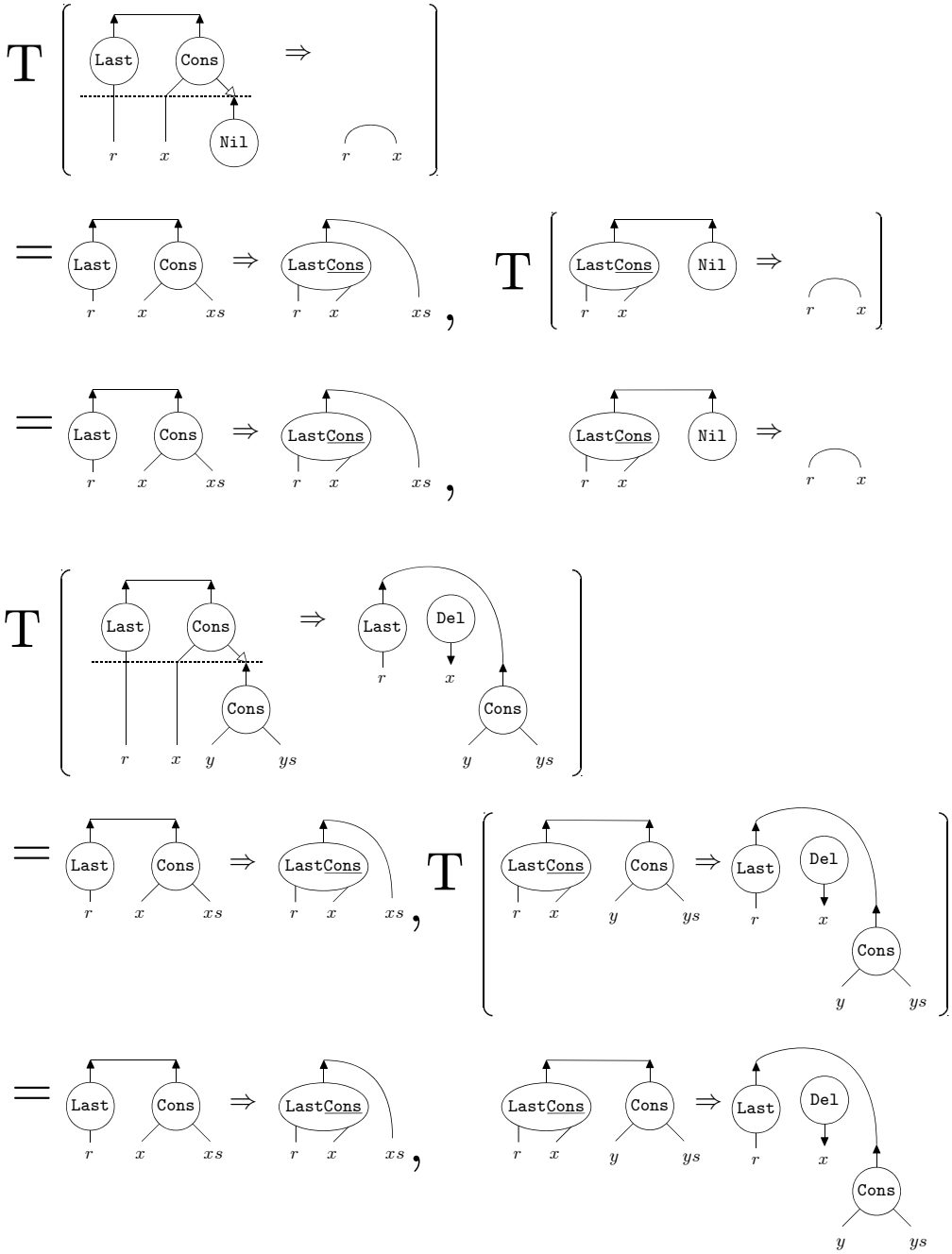



,




**Example 6.1.10**

*The rules in Example 6.1.6 are translated as follows:*



**Lemma 6.1.11**

*Let $\mathcal{R}$ be a well-formed rule set in* INP *and $R_1, R_2 \in \mathcal{R}$. Then, a rule set $\mathbf{T}[R_1] \cup \mathbf{T}[R_2]$ contains no rule such that $P \Rightarrow N_1$ and $P \Rightarrow N_2$ where $N_1 \neq N_2$.*

**Proof.** Let $R_1$, $R_2$ be $P_1 \Rightarrow M_1$, $P_2 \Rightarrow M_2$ respectively.

**case 1:** the active pairs in $P_1$ and $P_2$ are different. In this case, distinct names are introduced by $\mathbf{T}$ for those active pairs respectively. Therefore, every LHS of the rules generated by recursively applying $\mathbf{T}$ also have distinct active pairs.

**case 2:** the active pairs in $P_1$ and $P_2$ are the same. Because both $P_1$ and $P_2$ belong to the same sequential set, $P_1$ and $P_2$ have the same sequence of agents succeeding from the active pair. Therefore, in the set obtained from this sequence by using $\mathbf{T}$, there is no rule such that $P \Rightarrow M_1$ and $P \Rightarrow M_2$. For the remaining agents, it turns out that there is no such rule by applying case 1. □

**Proposition 6.1.12**

*Let $\mathcal{R}$ be a well-formed rule set in* INP*. The set $\bigcup \mathbf{T}[R]$ where $R \in \mathcal{R}$ is a proper rule set in* ORN*.*

**Proof.** From the definition of $\mathbf{T}$, it is clear that every LHS of rules obtained by using $\mathbf{T}$ contains only an active pair. Moreover, by Lemma 6.1.11, there is no rule $P \Rightarrow N_1$ and $P \Rightarrow N_2$, where $N_1 \neq N_2$, in the resulting rule set. □

**Proposition 6.1.13 (Conservativity)**

*Let $\mathcal{R}$ be a well-formed set of rules in* INP*. If $P \Rightarrow N \in \mathcal{R}$, then $P \rightarrow^* N$ by using the rules obtained by the translation $\mathbf{T}[P \Rightarrow N]$.*

**Proof.** If $P$ is just an active pair, then we can perform $P \rightarrow N$ because $\mathbf{T}[P \Rightarrow N] = P \Rightarrow N$.

If $P = \langle \alpha(\vec{x}) \bowtie \beta(\vec{y}, y),\ y - \gamma(\vec{z}),\ \vec{a} \rangle$ where $\vec{x}, \vec{y}, \vec{z}$ are sequences of auxiliary ports and $\vec{a}$ is a sequence of agents, then

$$\mathbf{T}[P \Rightarrow N] = (\alpha(\vec{x}) = \beta(\vec{y}, y) \Rightarrow \alpha\underline{\beta}(\vec{x}, \vec{y}) = y),\ \mathbf{T}[\left\langle \alpha\underline{\beta}(\vec{x}, \vec{y}) \bowtie \gamma(\vec{z}),\ \vec{a} \right\rangle \Rightarrow N].$$

By using the first rule,

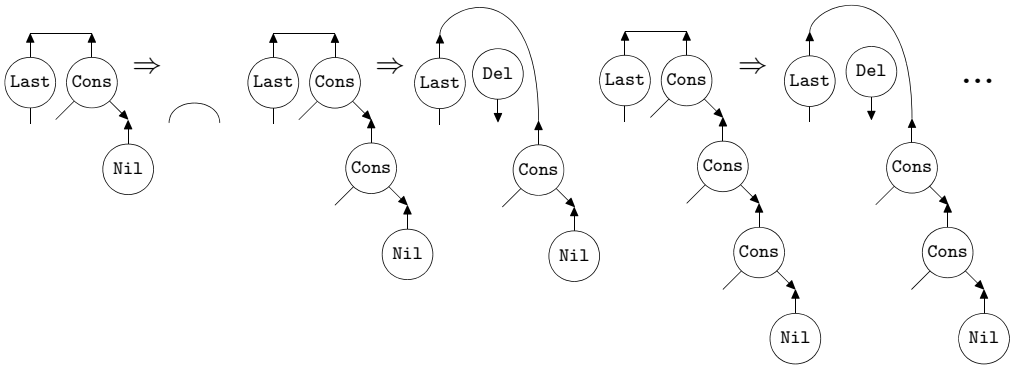
$$\alpha(\vec{x}) = \beta(\vec{y}, y),\ y = \gamma(\vec{z}) \ \rightarrow\ \alpha\underline{\beta}(\vec{x}, \vec{y}) = y,\ y = \gamma(\vec{z}) \ \rightarrow\ \alpha\underline{\beta}(\vec{x}, \vec{y}) = \gamma(\vec{z}).$$

Applying recursively this operation to the rule $\left\langle \alpha\underline{\beta}(\vec{x}, \vec{y}) \bowtie \gamma(\vec{z}),\ \vec{a} \right\rangle \Rightarrow N$ and the nested agent pair $\alpha\underline{\beta}(\vec{x}, \vec{y}) \bowtie \gamma(\vec{z})$, we will perform $P \rightarrow^* N$. □

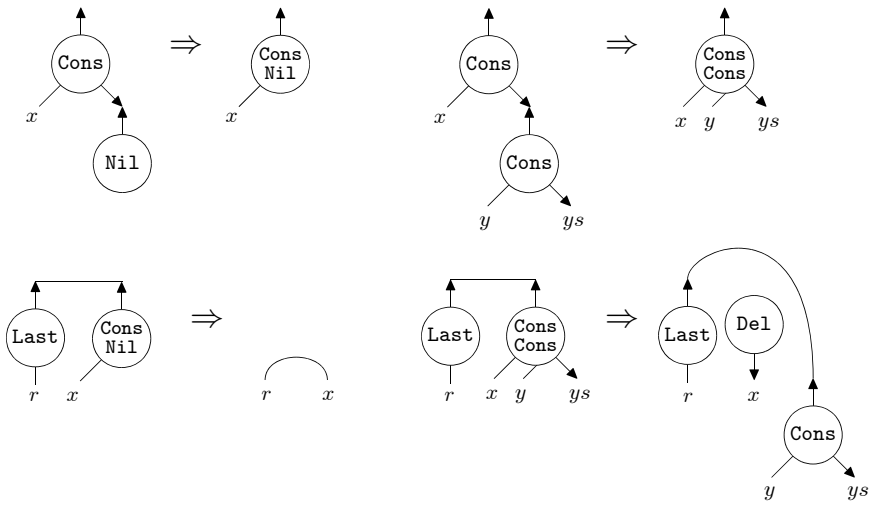
### 6.1.4 Related Works

In this section, we discuss other approaches to nested pattern matching by using methods that have been proposed as extensions of interaction nets.

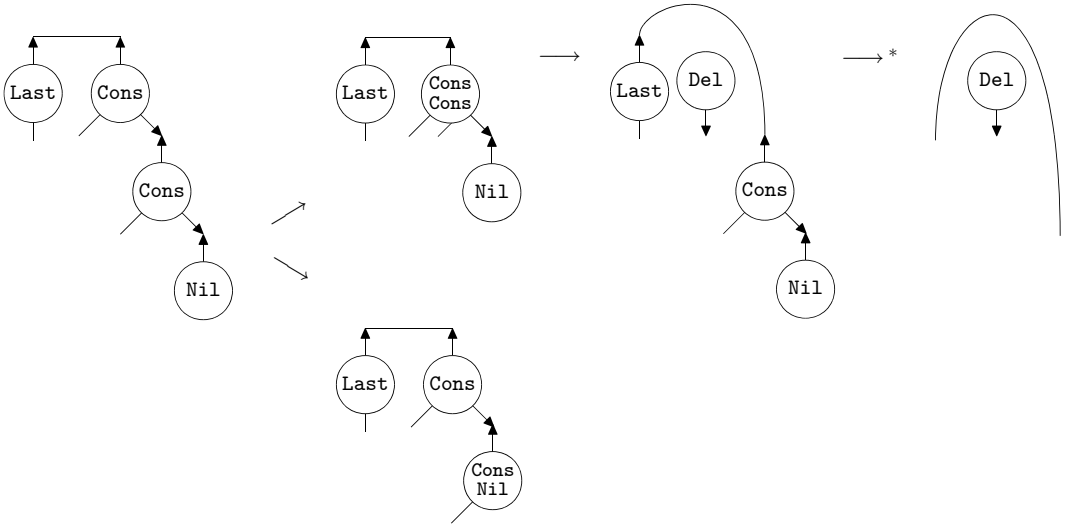
**Pattern matching on more than one argument** Sinot and Mackie [59] introduced *Macros* for interaction nets and they allow pattern matching on more than one argument by relaxing the restriction of one principal port per agent. Their system requires all principal ports of an agent in the LHS net of a rule to be connected to principal ports of other agents for the purpose of holding the property of strong confluence. Therefore, this system is useful as a conservative extension. However, we can hardly encode the function `Last` as it requires nested pattern matching. This is because in the case that the `Cons` agent has two principal ports, we have to write all cases as follows:



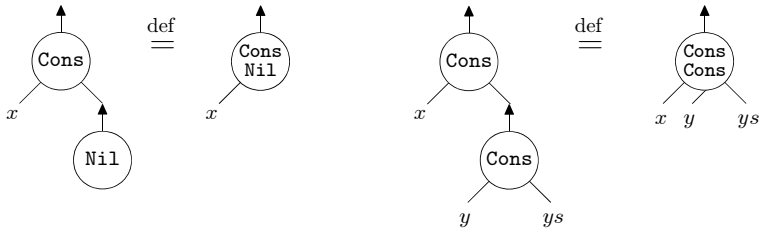

Alexiv's *interaction nets with multiple principal ports* (IMNPP) [2] is also useful for this purpose because this system also allows more than one principal port per agent. However, interactions are still performed only on an active pair. Therefore, in the case of nested pattern matching, we have to introduce auxiliary agents and rules as in Section 6.1.1. As another solution, we can introduce rules between `Cons` and `Nil`:



These cause, however, computation between the list structures even if it is not needed.




**Computation for nets** Bechet [10] proposed computation for nets on interaction rules as *abbreviations*, where nets are captured as an agent and reductions of the agent are realised by the rules corresponding to the computation of the net. As an example of applying this method to nested pattern matching, we consider our example function `Last`. One solution is to define the agent `Last` by using other agents that have already been defined. It is not simple to find a good combination with those agents. As another solution, we introduce abbreviations for list structures:



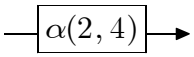

However, we have to define rules between `Last` and `Cons` for the case that those abbreviations are unfolded, therefore we have to introduce auxiliary agents in the end.

## 6.2 Agents and interaction rules with attributes

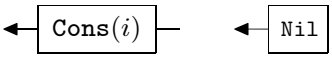
An agent that contains a natural number was introduced in the paper [19]. In this section we extend this notion so that any agent can have many attributes, which are integers. Next, we extend interaction rules so that these agents can be managed with arithmetic expressions and conditional operations.

### 6.2.1 Agents hold attributes

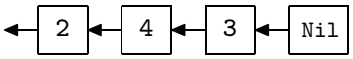
We write attributes in brackets after the symbols of agents: for instance, $\alpha(2, 4)$ is a node called $\alpha$ which holds the values 2 and 4. From now on, we draw agents not only as circles but also as squares or triangles.




Lists of integers are represented by using two agents: $\texttt{Cons}(i)$ and $\texttt{Nil}$ where $i$ is an attribute value:




To simplify the diagrams, we often just write the contents of the node and omit the name when no confusion will arise. For instance, a list of 2,4,3 is written as follows:




When agents in active pairs hold attributes, we also say that *the active pairs hold attributes*.

### 6.2.2 Interaction rules with expressions

In this section we extend interaction nets to create agents that hold new attributes as the result of applying arithmetic operations $o \in \{+, -, \times, \div, \texttt{mod}\}$ to some of attributes that are held by the original active pairs.

We consider relational operations $r \in \{=, \neq, <, \leq, >, \geq, \texttt{and}, \texttt{or}, \texttt{not}\}$ as arithmetic operations on integers, where the constant $\texttt{true}, \texttt{false}$ are denoted as $\texttt{1}, \texttt{0}$ respectively, and the results of relational operations are replaced with either 1 or 0. We call these operations *attribute operations*.

**Definition 6.2.1 (Interaction rules with expressions)**

- *Agents that hold expressions built on the attribute operations* $(+, -, \times, \div, \texttt{mod}, =, \neq, <, \leq, >, \geq, \texttt{and}, \texttt{or}, \texttt{not})$, *variables and integer numbers are called* agents with expressions. *For instance,* $\alpha((x + \texttt{1}) \times y, z)$ *is an agent with expressions.*

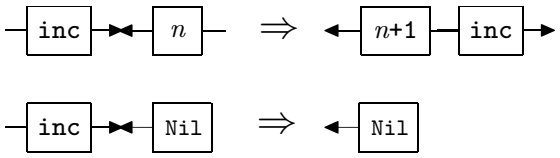
- Interaction rules with expressions *are defined as interaction rules such that*

  - *in the LHS, the active pair of the rule may hold variables for their attributes,*

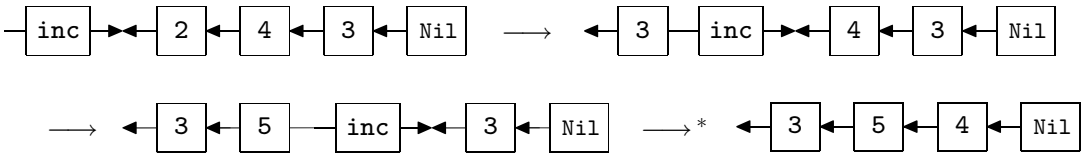
  - *in the RHS, agents with expressions may occur as long as every variable is held by the active pair in the LHS.*

- *Interaction rules with expressions may be applied to active pairs the same as in the original interaction nets. Attributes that are held by the active pairs of the rule are substituted for variables that occur in agents with expressions in the LHS of the rules, and each expression is performed and replaced with the calculation result. For instance, when a rule* $(\alpha(x_1, x_2), \beta(y)) \Rightarrow N$ *is applied to an active pair* $(\alpha(\mathtt{1}, \mathtt{2}), \beta(\mathtt{3}))$*, these* $x_1, x_2, y$ *that occur in* $N$ *are replaced with* $\mathtt{1}, \mathtt{2}, \mathtt{3}$ *respectively.*

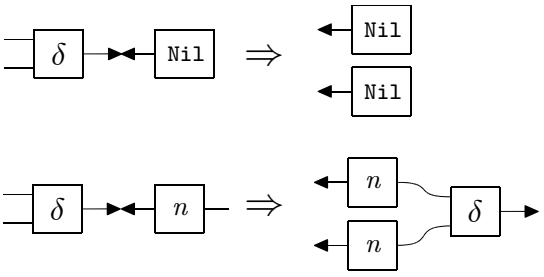
**Increments elements of lists** The following is an example of interaction rules with expressions such that the values of attributes in lists are incremented:



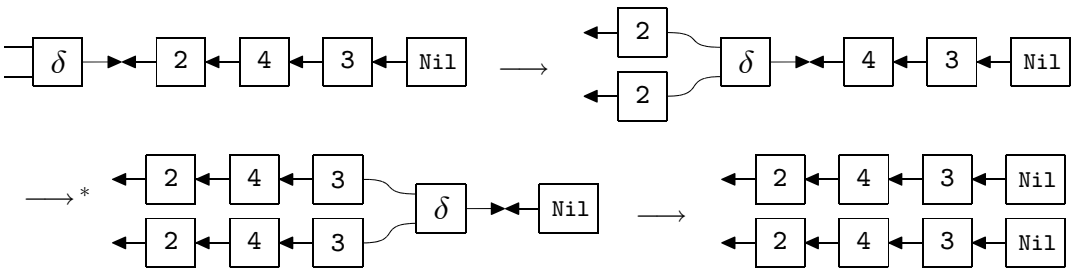

A list of 2,4,3 is changed into a list of 3,5,4 by applying the `inc` agent as follows:




**Duplication lists** The following is an example of interaction rules with expressions that duplicates lists:




A list is duplicated by applying the $\delta$ agent as follows:



### 6.2.3 Conditional interaction rules

We extended interaction rules with expressions so that these rules can be performed only when attributes in the active pairs meet given conditions.

**Definition 6.2.2 (Conditional interaction rules)**

- *Conditions for an active pair* $(\alpha(x_1, \ldots, x_n), \beta(y_1, \ldots, y_m))$ *are expressions built on attribute operations, integer numbers, and only variables* $x_1, \ldots, x_n, y_1, \ldots, y_m$. *For instance,* $(x \geq 1)$ `and` $(x \leq 10)$ *is a condition for an active pair* $(\alpha(x), \beta)$.

- *We define* conditional interaction rules *for active pairs* $(\alpha(x_1, \ldots, x_n), \beta(y_1, \ldots, y_m))$ *as interaction rules with expressions such that a condition Cond for the active pairs is labelled on arrows:*

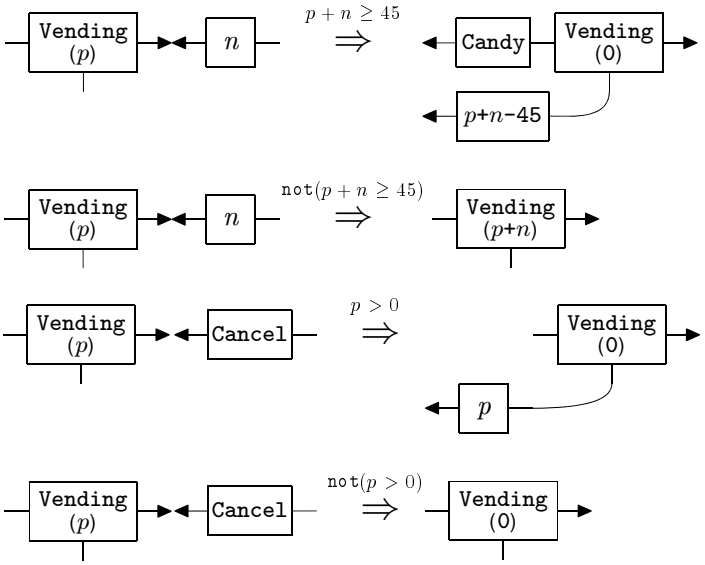
$$(\alpha(x_1, \ldots, x_n), \beta(y_1, \ldots, y_m)) \overset{Cond}{\Rightarrow} N$$

- *Conditional interaction rules may be applied to active pairs only if the evaluation result of the conditions is not* `0`.

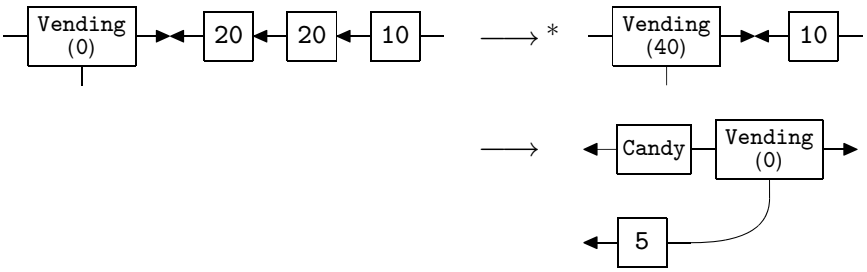
- *Each condition for the same active pair must be disjoint, and thus there is only one rule that can be applied to the active pair.*

We note that the Strong confluence property (Theorem 2.1.1) is preserved since there is only at most one conditional interaction rule for each active pair.

**Vending machine** This is an example of a vending machine such that a candy is sold at 45 pence. The agent `Vending` holds one integer number for the total value of inserted coins, and when the total value exceeds 45 pence, it outs one candy and changes. This logic is written by conditional interaction rules:

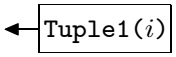


A list of coins 20p, 20p, 10p is changed into one candy and 5 pence:



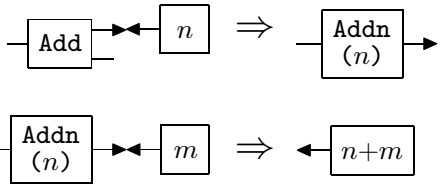

### 6.2.4 Examples

In this section we give some examples. Here we use the `Tuple1` agent to hold an integer number $i$:

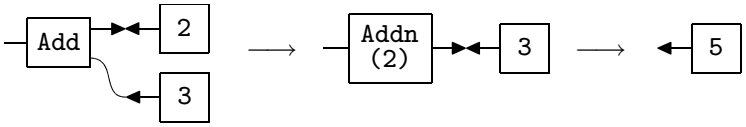



As long as there is no confusion, we omit the agent name and the bracket as follows:

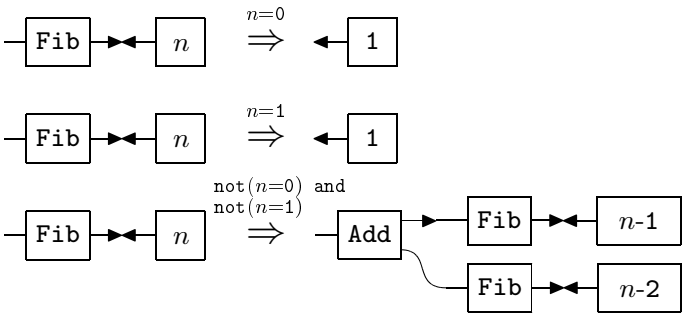

**Addition** The addition operation is written as the following rules:



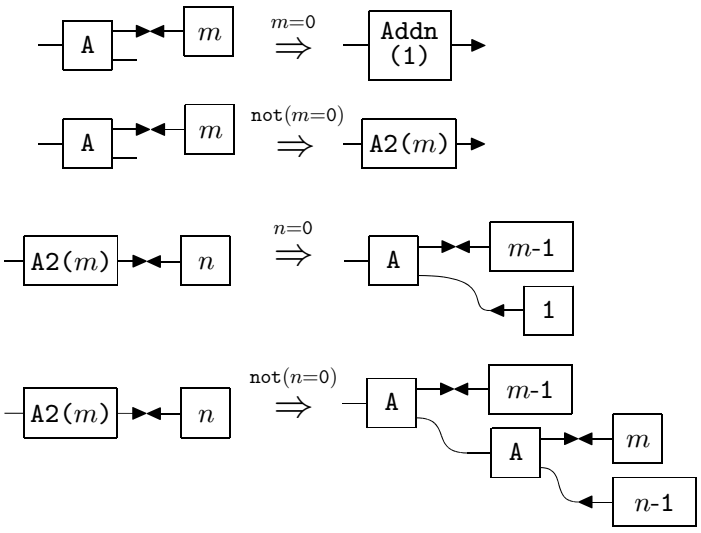

For instance, the computation result of $2 + 3$ is obtained as follows:



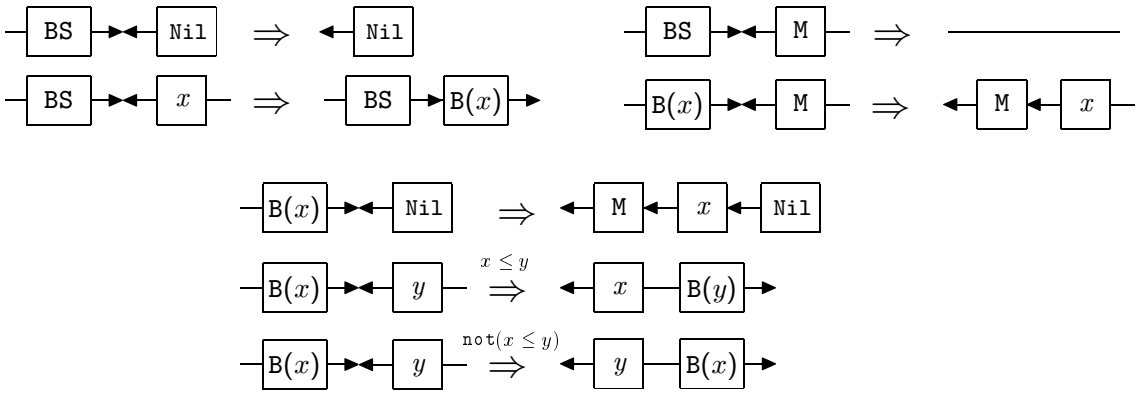

**Fibonacci number** The following is an example of interaction rules of Fibonacci number.



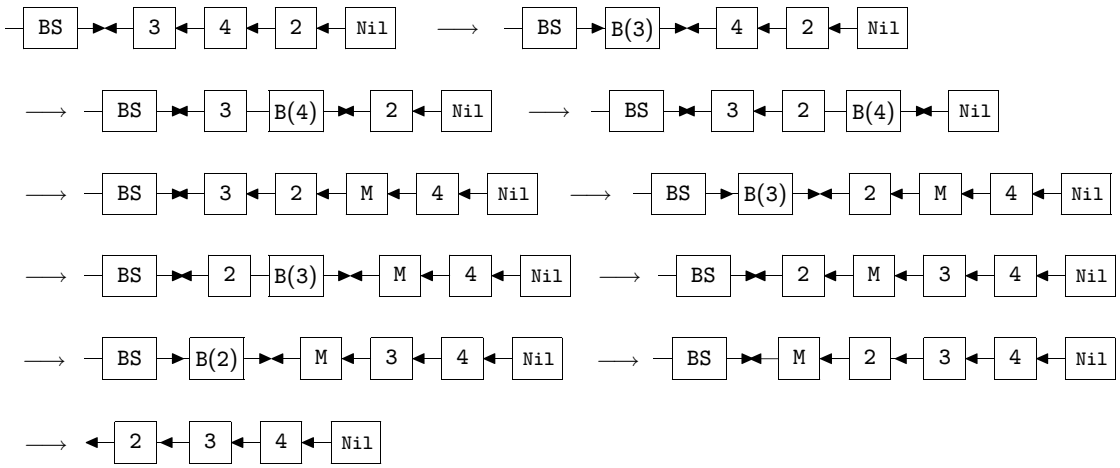
**Ackermann function** The following is an example of interaction rules of Ackermann function.



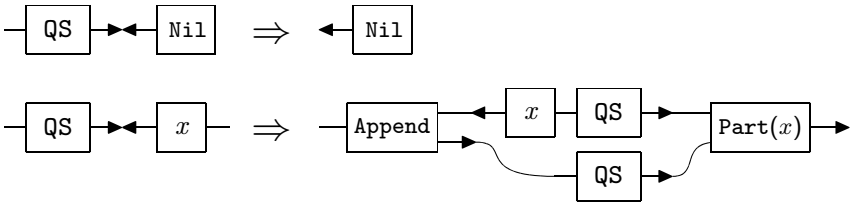

**Bubble Sort** The following is a set of rules for Bubble Sort algorithm. The agent `M` works as a separator to indicate that all elements after are sorted.



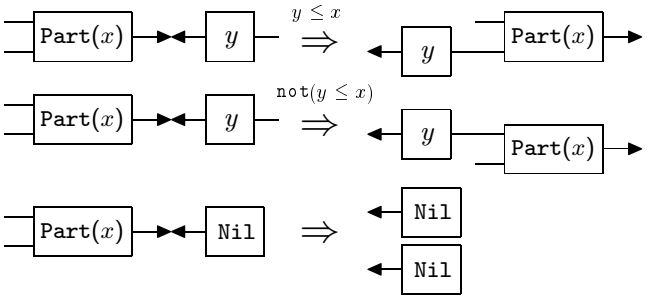

For instance, a list `[3,4,2]` is sorted as follows:



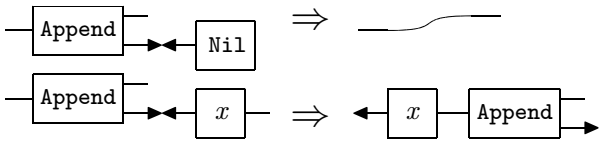
**Quick Sort** The next example is Quick Sort algorithm. The main mechanism of this algorithm is given by the following rules. The first element of the list is used as a pivot:



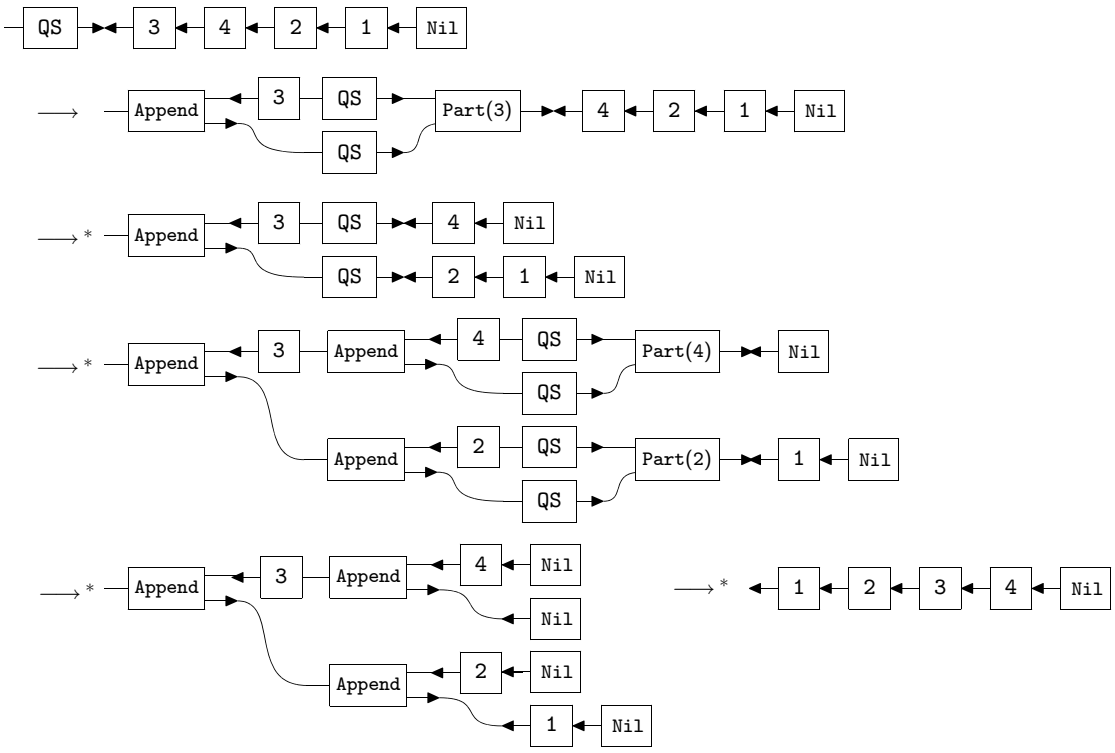

The agent `Part`$(x)$ splits a list into two ones by less than $x$ or not:




The agent `Append` appends given two lists into one:




For instance, a list `[3,4,2,1]` is sorted as follows:



## 6.3 Syntax

In this section we introduce new syntax to express the extensions for agents and interaction rules.

### 6.3.1 Nested pattern matching

Here we introduce the `case`-statements in the same style as functional programming languages in order to write interaction rules with nested pattern matching.

**Definition 6.3.1 (`case`-statements)**
*We define an abbreviation for sets of interaction rules of nested patterns with* `case`*-statements by the following grammar:*

$$\begin{aligned} P \Rightarrow\ & \texttt{case}\ x\ \texttt{of}\ \alpha_1(\vec{x}_1)\ \texttt{=>}\ E_1\ \texttt{|}\cdots\texttt{|}\ \alpha_n(\vec{x}_n)\ \texttt{=>}\ E_n \\ \stackrel{\text{def}}{=}\ & \{\langle P, x - \alpha_1(\vec{x}_1)\rangle \to E_1\} \cup \cdots \cup \{\langle P, x - \alpha_n(\vec{x}_n)\rangle \to E_n\} \end{aligned}$$

*where $P$ is a nested active pair, and $E_1, \ldots, E_n$, called case-expressions, are defined by the following grammar:*

$$\begin{aligned} E\ ::=\ & N \\ \mid\ & \texttt{case}\ x\ \texttt{of}\ \alpha_1(\vec{x}_1)\ \texttt{=>}\ E_1\ \texttt{|}\cdots\texttt{|}\ \alpha_n(\vec{x}_n)\ \texttt{=>}\ E_n. \end{aligned}$$

**Definition 6.3.2 (Sequential set for the `case`-statements)**
*Let $P \Rightarrow E$ be an abbreviation with* `case`*-statements. We define a set of nested active pairs by the following translation $\mathsf{Seq}(P, E)$ for nested agent pairs $P$ and the syntax of case-expressions $E$:*

$$\begin{aligned} &\mathsf{Seq}(\langle P\rangle, N) \stackrel{\text{def}}{=} \{\langle P\rangle\}, \\ &\mathsf{Seq}(\langle P\rangle, \texttt{case}\ x\ \texttt{of}\ \alpha_1(\vec{x}_1)\ \texttt{=>}\ E_1\ \texttt{|}\cdots\texttt{|}\ \alpha_n(\vec{x}_n)\ \texttt{=>}\ E_n) \\ &\qquad \stackrel{\text{def}}{=} \{\langle P\rangle\} \cup \mathsf{Seq}(\langle P, x - \alpha_1(\vec{x}_1)\rangle, E_1) \cup \cdots \cup \mathsf{Seq}(\langle P, x - \alpha_n(\vec{x}_n)\rangle, E_n). \end{aligned}$$

**Lemma 6.3.3**
*Let a set of rules $\alpha(\vec{x}) \bowtie \beta(\vec{y}) \Rightarrow E$ be given by an abbreviation with* `case`*-statements. Then $\mathsf{Seq}(\alpha(\vec{x}) \bowtie \beta(\vec{y}), E)$ is sequential.*

**Proof.** Let $\mathcal{P} = \mathsf{Seq}(\alpha(\vec{x}) \bowtie \beta(\vec{y}), E)$. First, we show that, for $\langle P,\ y_j - \gamma(\vec{z})\rangle \in \mathcal{P}$, $P \in \mathcal{P}$:

- By Definition 6.3.2, $\mathsf{Seq}(\langle P', y - \beta(\vec{y})\rangle, E')$ contains $P'$ and $\langle P', y - \beta(\vec{y})\rangle$. Every element in $\mathcal{P}$ occurs at the first argument of $\mathsf{Seq}$, and $\langle \alpha(\vec{x}) \bowtie \beta(\vec{y})\rangle \in \mathcal{P}$. Thus, for $\langle P,\ y_j - \gamma(\vec{z})\rangle \in \mathcal{P}$, $P \in \mathcal{P}$.

Next, we show that, when $\langle P,\ y_j - \gamma(\vec{z})\rangle \in \mathcal{P}$, then $\langle P,\ y - \alpha(\vec{w})\rangle \notin \mathcal{P}$ for all free ports $y$ in $P$ except the $y_j$ and for all agents $\alpha$:

- By Definition 6.3.2, the $\langle P,\ y_j - \gamma(\vec{z})\rangle$ is created by the case of a `case`-statement such as $\mathtt{Seq}(\langle P\rangle, \mathtt{case}\ y_j\ \mathtt{of}\ \gamma(\vec{z})\ \mathtt{=>}\ E\ \mathtt{|}\ \cdots\ \mathtt{|}\ \gamma'(\vec{z}')\ \mathtt{=>}\ E' \cdots)$. Other `case`-statements should occur in among $E$ and $E'$, so there is no element such that $\langle P,\ y - \alpha(\vec{w})\rangle \in \mathcal{P}$ where $y \neq y_j$. □

**Theorem 6.3.4**

*Let a rule set $\mathcal{R} = \alpha(\vec{x}) \bowtie \beta(\vec{y}) \Rightarrow E$ be given by an abbreviation with `case`-statements. Then $\mathcal{R}$ is well-formed.*

**Proof.** $\mathtt{Seq}(\alpha(\vec{x}) \bowtie \beta(\vec{y}), E)$ is a sequential set which contains every nested active pair of the LHS in $\mathcal{R}$. By Definition 6.3.2, for every rule $P \Rightarrow N$ in $\mathcal{R}$, there is no interaction rule $P' \Rightarrow N'$ in $\mathcal{R}$ such that $P'$ is a subnet of $P$. □

**Example 6.3.5**

*Interaction rules for Fibonacci number and Ackermann function are written as follows:*

$$
\begin{array}{rcl}
\mathtt{Fib}(r) = \mathtt{Z} & \Rightarrow & r = \mathtt{S}(\mathtt{Z}) \\
\mathtt{Fib}(r) = \mathtt{S}(x) & \Rightarrow & \mathtt{case}\ x\ \mathtt{of} \\
 & & \quad \mathtt{Z} \ \mathtt{=>}\ r = \mathtt{S}(\mathtt{Z}) \\
 & & \mathtt{|}\ \mathtt{S}(y) \ \mathtt{=>}\ y = \mathtt{Dup}(y_1, y_2),\ \mathtt{Fib}(r_1) = \mathtt{S}(y_1), \\
 & & \qquad \mathtt{Fib}(r_2) = y_2,\ \mathtt{Add}(r_2, r) = r_1 \\
\mathtt{A}(y, r) = \mathtt{Z} & \Rightarrow & r = \mathtt{S}(\mathtt{Z}) \\
\mathtt{A}(y, r) = \mathtt{S}(x) & \Rightarrow & \mathtt{case}\ y\ \mathtt{of} \\
 & & \quad \mathtt{Z} \ \mathtt{=>}\ \mathtt{Pred}(\mathtt{A}(\mathtt{S}(\mathtt{Z}), r)) = x \\
 & & \mathtt{|}\ \mathtt{S}(u) \ \mathtt{=>}\ u = \mathtt{Dup}(\mathtt{Pred}(\mathtt{A}(w, r)), \mathtt{A}(y, w))
\end{array}
$$

### 6.3.2 Agents and interaction rules with attributes

Here we introduce syntax for agents holding attributes and interaction rules with expressions and conditions.

**Agents hold attributes** We write attributes the same as auxiliary ports. For instance, $\alpha(\mathtt{2}, \mathtt{4}, r)$ means that the $\alpha$ agent holds two attributes $\mathtt{2}, \mathtt{4}$ and has one auxiliary port $r$.

In the case of agents with expressions, we also write variables the same as auxiliary ports. In order to distinguish those from auxiliary ports, we may use a modifier `int` before each attribute: For instance, $\mathtt{Cons}(\mathtt{int}\ x, r)$ means that the $x$ is used for an attribute variable and the $r$ is for an auxiliary port.

We extend the syntax for terms $t$ as follows:

$$\begin{array}{lll} V & ::= & \texttt{int}\ x \mid i \quad \text{where } i \text{ is an integer number,} \\ Arg & ::= & V \mid t, \\ t & ::= & x \mid \alpha(Arg_1, \ldots, Arg_n) \mid \$t. \end{array}$$

We use $\Delta, \Theta$ also for multisets of equations of those extended terms.

Definition 4.4.2 for name sets in terms is extended as follows:

**Definition 6.3.6 (Names in terms)**

*The set* $\mathsf{Name}(t)$ *of names of a term* $t$ *is defined in the following way, which extends to sequences of terms, equations, sequences of equations, and rules in the obvious way.*

$$\begin{array}{lll} \mathsf{Name}(i) & = & \emptyset \quad \textit{where } i \textit{ is an integer number,} \\ \mathsf{Name}(\texttt{(int )}x) & = & \emptyset, \\ \mathsf{Name}(x) & = & \{x\}, \\ \mathsf{Name}(\alpha(t_1, \ldots, t_n)) & = & \mathsf{Name}(t_1) \cup \cdots \cup \mathsf{Name}(t_n), \\ \mathsf{Name}(\$t) & = & \mathsf{Name}(t). \end{array}$$

Thus, the occurrence of attributes and attribute variables do not affect the linearity condition.

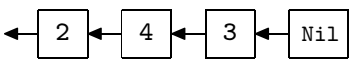
**Lists agents** List data-structures are common and useful to manage sequences of data. We introduce the $\texttt{Cons}(x, xs)$ and the `Nil` agents as built-in agents for lists and, write those by using the following abbreviations:

$$\texttt{[}x\texttt{|}xs\texttt{]}, \quad \texttt{[]}.$$

In addition, we introduce an abbreviation of $\texttt{[}x_1\texttt{|[}x_2\texttt{|[}x_3\texttt{|}\cdots\texttt{|Nil]]}\cdots\texttt{]}$ as

$$\texttt{[}x_1, x_2, x_3, \ldots\texttt{]}.$$

For instance, `[2,4,3]` denotes a list of 2,4,3:




**Tuple agents** The tuple data-structure is common to manage fixed-length data. In interaction nets programming, it is useful to concentrate on operations of attributes. We introduce the following $n$-tuple built-in agents:

$$\texttt{Tuple0}(), \ \texttt{Tuple1}(x_1), \ \texttt{Tuple2}(x_1, x_2), \ \texttt{Tuple3}(x_1, x_2, x_3), \ \ldots$$

and we write those agents by using the following abbreviations:

$$\texttt{()},\ \texttt{(}x_1\texttt{)},\ \texttt{(}x_1,x_2\texttt{)},\ \texttt{(}x_1,x_2,x_3\texttt{)},\ \ldots$$

For instance, the following is a 2-tuple for 3 and 5:

```
(3,5)
```

**Interaction rules with expressions** Each symbol of attribute operations

$$+,-,\times,\div,\texttt{mod},=,\neq,<,\leq,>,\geq,\texttt{and},\texttt{or},\texttt{not}$$

is written as

$$\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{mod},\texttt{==},\texttt{!=},\texttt{<},\texttt{<=},\texttt{>},\texttt{>=},\texttt{and},\texttt{or},\texttt{not}$$

respectively.

In the interaction rules with expressions, expressions $e_i$ that contain arithmetic operations are replaced with variable $v_i$, and written as $v_i = e_i$ in a `where`-clause as follows:

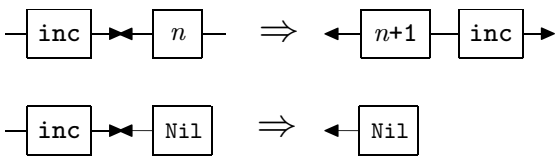
$$\alpha(x_1,\ldots,x_n) = \beta(y_1,\ldots,y_m) \Rightarrow \Delta\ \texttt{where}\ \ v_1 = e_1\ v_2 = e_2\ \cdots\ v_l = e_l$$

where

- variables for attributes among $x_1,\ldots,x_n,y_1,\ldots,y_m$ are written with the modifier `int`,
- every variable $v_i$ does not occur in $e_j$ such that $j < i$,
- each $e_i$ is an expression built on arithmetic operations, variables for the attributes in the LHS and variables $v_j$ such that $j < i$.

By this separation of expressions from equation sequences, operations for attributes are distinguished from the operations for the original interaction net.

Take the following rules as an example of this notation:




These are written as follows:

$$\begin{aligned}\texttt{inc}(r) = \texttt{[int}\ n\texttt{|}ns\texttt{]} &\Rightarrow r = \texttt{[}n_1\texttt{|}w\texttt{]}, \texttt{inc}(w) = ns\ \texttt{where}\ n_1 = n\texttt{+1}\\ \texttt{inc}(r) = \texttt{[]} &\Rightarrow r = \texttt{[]}\end{aligned}$$

**Conditional interaction rules** We introduce the following notation for conditional interaction rules for active pairs $P$ with conditions $Cond$ as follows:

$$P \overset{Cond}{\Rightarrow} \Delta$$

We also introduce the following notation for a set of conditional interaction rules for the same active pairs $P$ to ensure that conditions $Cond_i$ are disjoint:

$$\begin{array}{lll} P & | & Cond_1 \Rightarrow \Delta_1 \\ & | & Cond_2 \Rightarrow \Delta_2 \\ & \vdots & \\ & | & \texttt{otherwise} \Rightarrow \Delta_n \end{array}$$

as the following abbreviation:

$$\begin{cases} P \overset{Cond_1}{\Rightarrow} & \Delta_1 \\ P \overset{\texttt{not}(Cond_1)\ \texttt{and}\ Cond_2}{\Rightarrow} & \Delta_2 \\ \quad \vdots & \\ P \overset{\texttt{not}(Cond_1)\ \texttt{and}\ \cdots\ \texttt{and not}(Cond_{n-1})}{\Rightarrow} & \Delta_n \end{cases}$$

For instance, the rules for the candy vending machine are written as follows:

$$\begin{array}{l} \texttt{Vending}(\texttt{int}\ p, c, r) = \texttt{[int}\ n \texttt{|} ns\texttt{]} \\ \quad \texttt{|}\ p\texttt{+}n\texttt{>=45} \Rightarrow r = \texttt{candy}(w), \texttt{Vending}(\texttt{0}, c_1, w) = ns,\ c = \texttt{[}p_1 \texttt{|} c_1\texttt{]} \\ \qquad \texttt{where}\ p_1 = p\texttt{+}n\texttt{-45} \\ \quad \texttt{|}\ \texttt{otherwise} \Rightarrow \texttt{Vending}(p_1, c, w) = ns\ \texttt{where}\ p_1 = p\texttt{+}n \\ \texttt{Vending}(\texttt{int}\ p, c, r) = \texttt{cancel}(ns) \\ \quad \texttt{|}\ p\texttt{>0} \Rightarrow \texttt{Vending}(\texttt{0}, c_1, r) = ns, c = \texttt{[}p \texttt{|} c_1\texttt{]} \\ \quad \texttt{|}\ \texttt{otherwise} \Rightarrow \texttt{Vending}(\texttt{0}, c, r) = ns \end{array}$$

In the graph notation as well, we use the `otherwise` to denote negation of any other conditions.

**Example 6.3.7**

- *Interaction rules for Fibonacci numbers are written as follows:*

$$\begin{array}{rcl} \texttt{Add}(m, r) = \texttt{(int}\ n\texttt{)} & \Rightarrow & \texttt{Addn}(n, r) = m \\ \texttt{Addn}(\texttt{int}\ n, r) = \texttt{(int}\ m\texttt{)} & \Rightarrow & r = \texttt{(}i\texttt{)}\ \texttt{where}\ i = m + n \end{array}$$

$$\begin{array}{l} \texttt{Fib}(r) = \texttt{(int}\ a\texttt{)} \\ \quad \texttt{|}\ a\ \texttt{==}\ \texttt{0} \quad \Rightarrow\ r = \texttt{(1)} \\ \quad \texttt{|}\ a\ \texttt{==}\ \texttt{1} \quad \Rightarrow\ r = \texttt{(1)} \\ \quad \texttt{|}\ \texttt{otherwise} \ \Rightarrow\ \texttt{Fib}(\texttt{Add}(n, r)) = \texttt{(}b\texttt{)}, \texttt{Fib}(n) = \texttt{(}c\texttt{)} \\ \qquad\qquad \texttt{where}\ b\texttt{=}a\texttt{-1}\ c\texttt{=}a\texttt{-2} \end{array}$$

- *Ackermann function is written by the following rules:*

$$\texttt{A}(n, r) = \texttt{(int } m\texttt{)}$$
$$\texttt{ | } m \texttt{ == 0} \quad \Rightarrow \quad \texttt{Addn((1)}, r) = n$$
$$\texttt{ | otherwise} \quad \Rightarrow \quad \texttt{A2}(m, r) = n$$

$$\texttt{A2((int)}m, r) = \texttt{(int } n\texttt{)}$$
$$\texttt{ | } n \texttt{ == 0} \quad \Rightarrow \quad \texttt{A((1)}, r) = \texttt{(}m'\texttt{) where } m' = m\texttt{-1}$$
$$\texttt{ | otherwise} \quad \Rightarrow \quad \texttt{A}(w, r) = \texttt{(}m'\texttt{)}, \texttt{A((}n'\texttt{)}, w) = \texttt{(}m\texttt{)}$$
$$\texttt{where } m' = m\texttt{-1 } n' = n\texttt{-1}$$

*In the rules for Ackermann function, we can omit the declaration of* A2 *when we use the notation of nested pattern matching:*

$$\texttt{A}(n, r) = \texttt{(int } m\texttt{)}$$
$$\texttt{ | } m \texttt{ == 0} \quad \Rightarrow \quad \texttt{Addn((1)}, r) = n$$
$$\texttt{ | otherwise} \quad \Rightarrow \quad \texttt{case } n \texttt{ of}$$
$$\texttt{(int } n_1\texttt{) =>}$$
$$\texttt{ | } n_1 \texttt{ == 0} \quad \Rightarrow \quad \texttt{A((1)}, r) = \texttt{(}m'\texttt{) where } m' = m\texttt{-1}$$
$$\texttt{ | otherwise} \quad \Rightarrow \quad \texttt{A}(w, r) = \texttt{(}m'\texttt{)}, \texttt{A((}n'\texttt{)}, w) = \texttt{(}m\texttt{)}$$
$$\texttt{where } m' = m\texttt{-1 } n' = n_1\texttt{-1}$$

- *Evaluation results of* $F_{39}$ *and* $A(3, 8)$ *are obtained by equations* $\texttt{Fib}(r) = \texttt{(39)}$ *and* $\texttt{A((8)}, r) = \texttt{(3)}$ *respectively.*

**Example 6.3.8**

*The rule set of Bubble Sort is written as follows:*

$$\texttt{BS}(r) = \texttt{[]} \quad \Rightarrow \quad r = \texttt{[]}$$
$$\texttt{BS}(r) = \texttt{[}x\texttt{|}xs\texttt{]} \quad \Rightarrow \quad \texttt{B}(x, \texttt{BS}(r)) = xs$$
$$\texttt{BS}(r) = \texttt{M}(w) \quad \Rightarrow \quad r = w$$
$$\texttt{B(int } x, r) = \texttt{[]} \quad \Rightarrow \quad r = \texttt{M([}x\texttt{])}$$
$$\texttt{B(int } x, r) = \texttt{M}(w) \quad \Rightarrow \quad r = \texttt{M([}x\texttt{|}w\texttt{])}$$
$$\texttt{B(int } x, r) = \texttt{[int } y\texttt{|}ys\texttt{]}$$
$$\texttt{ | } x\texttt{<=}y \quad \Rightarrow \quad r = \texttt{[}x\texttt{|}w\texttt{]}, \texttt{B}(y, w) = ys$$
$$\texttt{ | otherwise} \quad \Rightarrow \quad r = \texttt{[}y\texttt{|}w\texttt{]}, \texttt{B}(x, w) = ys$$

**Example 6.3.9**

*The rule set of Quick Sort is written as follows:*

$$\texttt{QS}(r) = \texttt{[]} \quad \Rightarrow \quad r = \texttt{[]}$$
$$\texttt{QS}(r) = \texttt{[int } x\texttt{|}xs\texttt{]} \quad \Rightarrow \quad \texttt{Part}(x, \texttt{QS}(w), \texttt{QS(Append([}x\texttt{|}w\texttt{]}, r))) = xs$$

$$\texttt{Append}(a, b) = \texttt{[]} \quad \Rightarrow \quad a = b$$

$$\texttt{Append}(a, b) = \texttt{[}x\texttt{|}xs\texttt{]} \quad \Rightarrow \quad b = \texttt{[}x\texttt{|}w\texttt{]}, xs = \texttt{Append}(a, w)$$

$$\texttt{Part(int } x, a, b) = \texttt{[]} \quad \Rightarrow \quad a = \texttt{[]}, b = \texttt{[]}$$

$$\texttt{Part(int } x, a, b) = \texttt{[int } y\texttt{|}ys\texttt{]}$$

$$\texttt{|}\ y\texttt{<=}x \quad \Rightarrow \quad ys = \texttt{Part}(x, a, w), b = \texttt{[}y\texttt{|}w\texttt{]}$$

$$\texttt{|}\ \texttt{otherwise} \quad \Rightarrow \quad ys = \texttt{Part}(x, w, b), a = \texttt{[}y\texttt{|}w\texttt{]}$$

## 6.4 Extension of LL0

In this section we extend the syntax of LL0 and the compilation method so that agents and interaction rules with attributes can be expressed. Finally, we introduce an execution model.

### 6.4.1 Extension of the syntax of LL0

In this section, we extend the syntax of LL0.

**Attributes and expressions** We use attribute values the same as graph elements by putting a modifier `(int)` before the values, and thus an assignment of an attribute $i$ to a port $p \geq 1$ of a graph node $x$ is written as:

$$x[p] = \texttt{(int)}i$$

For instance, $x[1] = \texttt{(int)2}$ and $x[2] = \texttt{(int)10}$ are assignments of attribute variables `2` and `10` to ports $x[1]$ and $x[2]$ of a graph element $x$ respectively. Those values also can be assigned to variables by using the modifier `(int)` such as $v = \texttt{(int)2}$.

Attribute values that are assigned to variables can be referred in expressions by putting a modifier `(int)` before the variables. For instance, when an attribute value `2` is assigned to $v$, then $\texttt{(int)}v + \texttt{10}$ is an expression, and the evaluation result is `12`. With respect to ports of `L` and `R`, we only allow referring these in rule procedures such as `(int)L[1]`, `(int)R[2]` and so on.

Expressions, the same as attribute values, are also assigned to ports and variables by using the modifier `(int)`, and these are recognised as assignments of the calculation results of the expressions. For instance, when $x[1]$ has an attribute value `2`, then $x[2] = \texttt{(int)}(\texttt{(int)}x[1]\texttt{+1})$ is an assignment of the calculation result $x[1]$`+1`, thus `3`, to the port $x[2]$.

The following is the summary for the extension for instructions:

| Instruction | Description |
|---|---|
| $x[p]$ = `(int)`$e$ | Assign the result of an attribute expression $e$ to a port $p \geq 1$ of a graph element $x$. |
| $v$ = `(int)`$e$ | Assign the result of an attribute expression $e$ to a variable $v$. |

and for expressions:

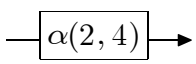
| Expression | Description |
|---|---|
| `(int)L`$[p]$ | Deal with the port $p$ of the left agent of the operated active pair as an attribute value. |
| `(int)R`$[p]$ | Deal with the port $p$ of the right agent of the operated active pair as an attribute value. |
| `(int)`$v$ | Deal with the value of the variable $v$ as an attribute value. |
| $op\ e_1$ | Apply an unary operation $op \in \{$`-`, `not`$\}$ to an expression $e_1$. |
| $e_1\ op\ e_2$ | Apply a binary operation $op$ to expressions $e_1, e_2$ where $op \in \{$`+`, `-`, `*`, `/`, `mod`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `and`, `or`$\}$ |

The syntax is extended as follows (where original definitions are underlined):

$$
\begin{aligned}
\underline{\langle operation \rangle} &::= \langle attrAssign \rangle \mid \underline{\langle assignment \rangle \mid \langle disposeAgent \rangle \mid \langle opEquation \rangle} \\
\langle attrAssig \rangle &::= \langle var \rangle\ '\texttt{=}'\ \texttt{(int)}\ \langle attrExp \rangle \\
\langle attrExp \rangle &::= \langle integer \rangle \mid \texttt{(int)}\ \langle var \rangle \mid \texttt{(int)}\ (' \texttt{L}' \mid ' \texttt{R}')\ '[' \ \langle num \rangle\ ']' \mid \\
&\quad \langle unaryArith \rangle \mid \langle binaryArith \rangle\ |'\ (' \ \langle attrExp \rangle\ ')' \\
\langle unaryArith \rangle &::= (' \texttt{-}' \mid ' \texttt{not}') \mid \langle attrExp \rangle \\
\langle binaryArith \rangle &::= \langle attrExp \rangle \mid (' \texttt{+}' \mid ' \texttt{-}' \mid ' \texttt{*}' \mid ' \texttt{/}' \mid ' \texttt{mod}' \mid \\
&\quad ' \texttt{==}' \mid ' \texttt{!=}' \mid ' \texttt{<}' \mid ' \texttt{<=}' \mid ' \texttt{>}' \mid ' \texttt{>=}' \mid ' \texttt{and}' \mid ' \texttt{or}') \mid \langle attrExp \rangle \\
\langle integer \rangle &::= '-' \ \langle num \rangle \mid \langle num \rangle
\end{aligned}
$$

**Example 6.4.1**

*An agent holding attributes*




*is allocated by the following instructions:*

```
aAlpha = mkAgent(ALPHA)
aAlpha[1] = (int)2
aAlpha[2] = (int)4
```

**Example 6.4.2**

*A list* `[2,4,3]` *is allocated by the following instructions:*

```
// []
aNil=mkAgent(NIL)
// [3]
aCons1=mkAgent(CONS)
aCons1[1]=(int)3
aCons1[2]=aNil
// [4,3]
aCons2=mkAgent(CONS)
aCons2[1]=(int)4
aCons2[2]=aCons1
// [2,4,3]
aCons3=mkAgent(CONS)
aCons3[1]=(int)2
aCons3[2]=aCons2
```

**Interaction rules** We introduce `if`-statements to the syntax of rule procedures so that conditional rules can be described as follows:

```
rule α β {
  if (Cond_1) {···}
  elif (Cond_2) {···}
    ⋮
  elif (Cond_n) {···}
  else {···}
}
```

where $Cond_i$ are conditions. The same as the block in rule procedures, we also write instructions between { and } in `if`-statements, which are called `if`-statement blocks, and variables introduced in an `if`-statement block can live only in the block.

In the execution of an `if`-statement, each condition $Cond_i$ is evaluated in the order $Cond_1, Cond_2, \ldots$ until its evaluation result is `0`. When the evaluation result of $Cond_j$ is not `0`, then the block placed at the right of the $Cond_j$ is evaluated as the execution of the `if`-statement. Otherwise, the block of `else` is evaluated. These `elif`-clauses can be omitted if those are not needed.

With respect to the syntax of nested pattern matching, we deal with it as a syntax sugar, and thus we assume that the `case`-statement is translated into normal interaction

rules automatically.

The syntax for rules are extended as follows:

$$
\begin{aligned}
\langle ruleBlock \rangle &::= \langle ifClause \rangle\ \langle elifClause \rangle^*\ \langle elseClause \rangle \mid \underline{'\{'\ \langle operation \rangle^*\ '\}'} \\
\langle ifClause \rangle &::= \texttt{'if'}\ \texttt{'('}\ \langle attrExp \rangle\ \texttt{')'}\ \langle ifBlock \rangle \\
\langle elifClause \rangle &::= \texttt{'elif'}\ \texttt{'('}\ \langle attrExp \rangle\ \texttt{')'}\ \langle ifBlock \rangle \\
\langle elseClause \rangle &::= \texttt{'else'}\ \langle ifBlock \rangle \\
\langle ifBlock \rangle &::= \texttt{'\{'}\ \langle operation \rangle^*\ \texttt{'\}'}
\end{aligned}
$$

**Example 6.4.3**

*The interaction rule for Fibonacci number is written as follows:*

```
rule Fib Tuple1 {
  stackFree()
  if ((int)R[1]==0) {
    aTP1 = mkAgent(Tuple1)
    aTP1[1] = (int)1
    push(L[1],aTP1)
  } elif ((int)R[1]==1) {
    aTP1 = mkAgent(Tuple1)
    aTP1[1] = (int)1
    push(L[1],aTP1)
  } else {
    b = (int)((int)R[1]-1)
    c = (int)((int)R[1]-2)
    w1 = mkName()
    aFib = mkAgent(Fib)
    aAdd = mkAgent(Add)
    aAdd[1]=w1
    aAdd[2]=L[1]
    aFib[1] = aAdd
    aTP1 = mkAgent(Tuple1)
    aTP1[1] = b
    push(aFib,aTP1)
    bFib = mkAgent(Fib)
    bFib[1] = w1
    bTP1 = mkAgent(Tuple1)
```

```
    bTP1[1] = c
    push(bFib,bTP1)
  }
  free(L)
  free(R)
}
```

### 6.4.2 Extension of the compilation method

In this section we extend the compilation method that is defined by Definition 5.2.2 and 5.2.4 into the extended syntax. First we extend the compilation of terms.

- The translation $\mathsf{Compile}_t$ defined by Definition 5.2.2 is extended into integer numbers as follows:

$$\mathsf{Compile}_t(n) \stackrel{\text{def}}{=} (\text{“”}, \text{“}\texttt{(int)}\{n\}\text{”}) \quad \text{where } n \text{ is an integer number}$$

Next we extend the compilation for rules.

- The translation $\mathsf{Compile}_r$ defined by Definition 5.2.4 is re-defined as follows:

$$\begin{array}{lcl}
\mathsf{Compile}_r(\alpha(\vec{x}) = \beta(\vec{y}) \Rightarrow \Theta) & \stackrel{\text{def}}{=} & \mathsf{Compile}_r(\alpha(\vec{x}) = \beta(\vec{y}) \Rightarrow \Theta \ \texttt{where}\ ) \\
\mid \mathsf{Compile}_r(\alpha(\vec{t}) = \beta(\vec{s}) \Rightarrow \Theta & \stackrel{\text{def}}{=} & \text{“}\texttt{rule}\ \{\alpha\}\ \{\beta\}\ \{\text{”} \\
\quad \texttt{where}\ \ v_1 = e_1 \cdots v_n = e_n) & & + \text{“}\texttt{stackFree()}\text{”} \\
& & + \mathsf{Compile}_{Rinst}(\alpha(\vec{t}) = \beta(\vec{s}) \Rightarrow \Theta \\
& & \quad \texttt{where}\ \ v_1 = e_1 \cdots v_n = e_n) \\
& & + \text{“}\texttt{free(L)}\text{”} \\
& & + \text{“}\texttt{free(R)}\text{”} \\
& & + \text{“}\}\text{”} \\
\mid \mathsf{Compile}_r(\alpha(\vec{t}) = \beta(\vec{s}) & \stackrel{\text{def}}{=} & \text{“}\texttt{rule}\ \{\alpha\}\ \{\beta\}\ \{\text{”} \\
\quad \mid e_1 \Rightarrow \Theta_1 & & + \text{“}\texttt{stackFree()}\text{”} \\
\quad \mid e_2 \Rightarrow \Theta_2 & & + \text{“}\texttt{if}\ \ (\text{”} + \mathsf{Compile}_{Rexp}(e_1) + \text{“})\ \ \{\text{”} \\
\quad \vdots & & + \ \mathsf{Compile}_{Rinst}(\alpha(\vec{t}) = \beta(\vec{s}) \Rightarrow \Theta_1) + \text{“}\}\text{”} \\
\quad \mid \texttt{otherwise} \Rightarrow \Theta_n) & & + \text{“}\texttt{elif}\ \ (\text{”} + \mathsf{Compile}_{Rexp}(e_2) + \text{“})\ \ \{\text{”} \\
& & + \ \mathsf{Compile}_{Rinst}(\alpha(\vec{t}) = \beta(\vec{s}) \Rightarrow \Theta_2) + \text{“}\}\text{”} \\
& & \quad \vdots
\end{array}$$

$$
\begin{array}{l}
\quad + \text{“}\texttt{else } \{\text{”} \\
\quad + \ \mathsf{Compile}_{Rinst}(\alpha(\vec{t}) = \beta(\vec{s}) \Rightarrow \Theta_n) + \text{“}\}\text{”} \\
\quad + \text{“}\texttt{free(L)}\text{”} \\
\quad + \text{“}\texttt{free(R)}\text{”} \\
\quad + \text{“}\}\text{”};
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{Compile}_{Rinst}(\alpha(\vec{t}) = \beta(\vec{s}) \Rightarrow \Theta) & \stackrel{\text{def}}{=} & \mathsf{Compile}_{Rinst}(\alpha(\vec{t}) = \beta(\vec{s}) \Rightarrow \Theta \ \texttt{where}) \\
\quad | \ \mathsf{Compile}_{Rinst}(\alpha(\vec{t}) = \beta(\vec{s}) \Rightarrow \Theta & \stackrel{\text{def}}{=} & \mathsf{let} \\
\qquad \texttt{where } \ v_1 = e_1 \cdots v_n = e_n) & & \qquad \vec{x} = \mathsf{remInt}(\vec{t}); \quad \vec{y} = \mathsf{remInt}(\vec{s}); \\
& & \qquad \mathrm{N}_1 = \mathsf{Compile}_{rn}(\vec{x}, \mathtt{L}, \emptyset); \\
& & \qquad \mathrm{N}_2 = \mathsf{Compile}_{rn}(\vec{y}, \mathtt{R}, \mathrm{N}_1); \\
& & \qquad (\mathrm{N}, c_1) = \mathsf{makeN}'( \\
& & \qquad\quad \mathsf{Name}(\Theta) - \{\vec{x}, \vec{y}\}, \mathrm{N}_2); \\
& & \quad \mathsf{in} \\
& & \qquad c_1 + \mathsf{Compile}_{Res}(\Theta, \\
& & \qquad\quad v_1 = e_1, \ldots, v_n = e_n); \\
& & \quad \mathsf{end};
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{remInt}(\texttt{(int)}x, \vec{t}) & \stackrel{\text{def}}{=} & (x, \mathsf{remInt}(\vec{t})) \\
\quad | \ \mathsf{remInt}(t, \vec{t}) & \stackrel{\text{def}}{=} & (t, \mathsf{remInt}(\vec{t}));
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{Compile}_{Res}(\Theta, & \stackrel{\text{def}}{=} & \mathsf{let} \\
\quad v_1 = e_1, \ldots, v_n = e_n) & & \quad \mathrm{N}[v_1] := \mathsf{freshStr}(); \cdots \mathrm{N}[v_n] := \mathsf{freshStr}(); \\
& & \quad c_1 = \text{“}\{\mathrm{N}[v_1]\}\texttt{=(int)(}\{\mathsf{Compile}_{exp}(e_1)\}\texttt{)}\text{”}; \\
& & \qquad \vdots \\
& & \quad c_n = \text{“}\{\mathrm{N}[v_n]\}\texttt{=(int)(}\{\mathsf{Compile}_{exp}(e_n)\}\texttt{)}\text{”}; \\
& & \mathsf{in} \\
& & \quad c_1 + \cdots + c_n + \mathsf{Compile}_{es}(\Theta) \\
& & \mathsf{end};
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{Compile}_{Rexp}(i) & \stackrel{\text{def}}{=} & i \quad \text{where } i \text{ is an integer number} \\
\mid \mathsf{Compile}_{Rexp}(x) & \stackrel{\text{def}}{=} & \text{“}\texttt{(int)}\{\mathrm{N}[x]\}\text{”} \\
\mid \mathsf{Compile}_{Rexp}(op\ e) & \stackrel{\text{def}}{=} & op + \mathsf{Compile}_{Rexp}(e) \quad \text{where } op \in \{\texttt{-}, \texttt{not}\} \\
\mid \mathsf{Compile}_{Rexp}(e_1\ op\ e_2) & \stackrel{\text{def}}{=} & \mathsf{Compile}_{Rexp}(e_1) + op + \mathsf{Compile}_{Rexp}(e_2) \\
 & & \quad \text{where } op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{mod}, \\
 & & \quad\quad \texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{and}, \texttt{or}\} \\
\mid \mathsf{Compile}_{Rexp}((e)) & \stackrel{\text{def}}{=} & \mathsf{Compile}_{Rexp}(e);
\end{array}
$$

**Example 6.4.4**

*Here, as an example, we take the interaction rule for Fibonacci number in Example 6.3.7:*

$$
\begin{array}{l}
\texttt{Fib}(r) = \texttt{(int } a\texttt{)} \\
\quad \mid a \texttt{ == 0} \quad \Rightarrow \quad r = \texttt{(1)} \\
\quad \mid a \texttt{ == 1} \quad \Rightarrow \quad r = \texttt{(1)} \\
\quad \mid \texttt{otherwise} \quad \Rightarrow \quad \texttt{Fib(Add}(n, r)) = \texttt{(}b\texttt{)}, \texttt{Fib}(n) = \texttt{(}c\texttt{)} \\
\qquad\qquad\qquad\qquad\quad \texttt{where } b\texttt{=}a\texttt{-1 } c\texttt{=}a\texttt{-2}
\end{array}
$$

*By applying* $\mathsf{Compile}_r$ *to the rule, the following is obtained:*

```
"rule Fib Tuple1 {
   StackFree()
   if ({Compile_Rexp(a == 0)}) {
     {Compile_Rinst(Fib(r) = (int a) ⇒ r = (1))} }
   elif ({Compile_Rexp(a == 1)}) {
     {Compile_Rinst(Fib(r) = (int a) ⇒ r = (1))} }
   else { {Compile_Rinst(Fib(r) = (int a) ⇒ Θ where b=a-1 c=a-2)} }
   free(L)
   free(R)
 }"
```

*where* $\Theta$ *is* $\texttt{Fib(Add}(n, r)) = \texttt{(}b\texttt{)}, \texttt{Fib}(n) = \texttt{(}c\texttt{)}$.

*We manage the first* `if`*-block. By unfolding* $\mathsf{Compile}_{Rexp}(a \texttt{ == 0})$ *the following is obtained:*

```
(int)R[1]==0
```

*By unfolding* $\mathsf{Compile}_{Rinst}(\texttt{Fib}(r) = \texttt{(int } a\texttt{)} \Rightarrow r = \texttt{(1)})$*, the following is obtained:*

```
aTuple1=mkAgent(Tuple1)
aTuple1[1]=(int)1
push(L[1],aTuple1)
```

*Next we manage the last* `if`*-block* $\mathsf{Compile}_{Rinst}(\texttt{Fib}(r) = \texttt{(int } a\texttt{)} \Rightarrow \Theta)$. *This is unfolded as follows:*

```
n=mkName()
Compile_Res(Θ, b=a-1, c=a-2)
```

*with the following name table* N*:*

$$\mathrm{N} = \{(n, \mathtt{n})\}.$$

*The* $\mathsf{Compile}_{Res}(\Theta, b{=}a{-}1, c{=}a{-}2)$ *is unfolded as follows:*

```
b=(int)((int)R[1]-1)
c=(int)((int)R[1]-2)
Compile_es(Θ)
```

*Therefore, the outlook of the compilation result is obtained as follows:*

```
rule Fib Tuple1 {
  StackFree()
  if ((int)R[1]==0) {
    aTuple1=mkAgent(Tuple1)
    aTuple1[1]=(int)1
    push(L[1],aTuple1)
  } elif ((int)R[1]==1) {
    aTuple1=mkAgent(Tuple1)
    aTuple1[1]=(int)1
    push(L[1],aTuple1)
  } else {
    n=mkName()
    b=(int)((int)R[1]-1)
    c=(int)((int)R[1]-2)
    Compile_es(Θ)
  }
  free(L)
  free(R)
}
```

*Taking account of the unfolding result of* $\mathsf{Compile}_{es}(\Theta)$, *this has the same operations of the rule procedure in Example 6.4.3.*

## 6.5 Extension of execution models

In this section, we explain how these extended nets are evaluated. For this purpose, we extend the execution model based on the standardised implementation model in the C language. Next, we show correspondence of extended codes in LL0 with ones in the C language. We also extend the byte-code machine to evaluate those extended nets.

### 6.5.1 Data-structures for agents, ports and attributes

Attributes, which are integer numbers, are held by agents. To manage attributes the same as ports, thus to incorporate ports and integer numbers, we introduce VALUE type [6], which is used in the implementation of Ruby [60]:

```
typedef unsigned long VALUE;
```

Thus, every object is managed by a pointer such as `void*`, which is assumed equivalent to `unsigned long`, and referred by casting the pointer. To prepare a common method to recognise which sort of a given `VALUE` object, the new `Agent` and `Name` structures have the following `Basic` structure that has the original `id` value as the first element:

```
typedef struct {
  int id;
} Basic;
```

Thus, the new data-structures for those nodes are written as follows:

```
typedef struct {
  Basic basic;
  VALUE port[MAX_PORT];
} Agent;
typedef struct {
  Basic basic;
  VALUE port;
} Name;

#define RBASIC(a) ((Basic *)(a))
#define RAGENT(a) ((Agent *)(a))
#define RNAME(a) ((Name *)(a))
```

For a given `a1` of the `VALUE` type, the `id` can be referred by `RBASIC(a1)->id`, and according to the `id`, other elements can be referred by `RAGENT(a1)` and `RNAME(a1)`.

Integer numbers are embedded into values of pointers [25] by using arithmetic shift operations, taking advantage of the alignment of pointers such as a 4-byte boundary. In this implementation, we restrict integer numbers to 31-bits representation, called *fixed numbers*, and we embed those into values of `VALUE` by using the lowest bit of pointers as a tag of the fixed numbers [6]:

```
#define FIXNUM_FLAG 0x01
#define INT2FIX(i) ((VALUE)(((long)(i) << 1) | FIXNUM_FLAG))
#define FIX2INT(i) ((int)(i) >> 1)
#define IS_FIXNUM(i) ((VALUE)(i) & FIXNUM_FLAG)
```

By introducing the `VALUE` type, the following basic functions are also changed:

```
VALUE mkAgent(int id);
VALUE mkName();
VALUE mkInd();
void pushActive(VALUE a1, VALUE a1);
int popActive(VALUE *a1, VALUE *a1);
```

### 6.5.2 Execution model in the C language

In this section, we explain how these extended nets are evaluated, showing correspondence of codes in LL0 with ones in the C language.

**Instructions** The correspondence of the extended instructions is as follows:

- $x[p] =$ `(int)`$e$

  The calculation result of the expression $e$ is managed as a fixed number, and thus this corresponds to the following code:

  $x$`->port[`$p$`-1]=INT2FIX(`$e$`);`

- $v =$ `(int)`$e$

  The evaluation result of the expression $e$ is assigned into $v$ as a fixed number, and thus this corresponds to the following code:

  $v$`=INT2FIX(`$e$`);`

When the $v$ occurs at first, then we add a declaration for the $v$:

```
VALUE v=INT2FIX(e);
```

Next, we show the correspondence of expressions:

- `(int)L`$[p]$

  This refers the port $x[p]$ of the left agent of the operated active pair as a fixed number, and thus this corresponds to the following code:

  ```
  FIX2INT(RAGENT(a1)->port[p − 1])
  ```

- `(int)R`$[p]$

  The same as the case of `(int)L`$[p]$, this corresponds to the following code:

  ```
  FIX2INT(RAGENT(a2)->port[p − 1])
  ```

- `(int)`$v$

  This also refers the value of the variable $v$ as a fixed number, and thus this corresponds to the following code:

  ```
  FIX2INT(v)
  ```

- $op\ e_1,\ e_1\ op\ e_2$

  We have the straightforward correspondences in those expressions since the C language has almost the same operations, except for `not`, `and`, `or` that correspond to `!`, `&&`, `||` respectively.

**Rule procedures** The extension of rule procedures is the `if`-statement that is written as follows:

```
rule α β {
  if (Cond1) {···}
  elif (Cond2) {···}
    ⋮
  elif (Condn) {···}
  else {···}
}
```

Each expression $Cond_i$ is represented as an expression in the C language by applying `FIX2INT` to variables and ports, and we call it $Cond'_i$. Then, the C language also has the `if`-statement that works the same as the definition of the `if`-statement in LL0, and thus the above rule procedure corresponds to the following function in the C language:

```
void α_β(VALUE a1, VALUE a2) {
  if (Cond'_1) {···}
  elif (Cond'_2) {···}
    ⋮
  elif (Cond'_n) {···}
  else {···}
}
```

The run-time function `eval` is also changed by using the type `VALUE` as follows:

```
void eval() {
  VALUE a1, a2;
  while (popActive(&a1, &a2)) {
    if (RBASIC(a2)->id != ID_NAME) {
      if (RBASIC(a1)->id != ID_NAME) {
        R[RBASIC(a1)->id][RBASIC(a2)->id](a1,a2);
      } /* The below is operations for x=t */
          ⋮
      }
    } else {
      /* operations for t=y and x=y */
        ⋮
    }
  }
}
```

### 6.5.3 Execution model in the byte-code interpreter

To evaluate expressions and the extended rule procedures, we add codes into the set of byte-codes as shown in Figure 6.1. The code `JMPEQ0` is used to manage the program counter according to evaluation results of conditional expressions, and the other codes are used to operate on expressions.

| Byte-code | Description |
|---|---|
| `MKVAL A B` | `Reg(A):=INT2FIX(B)` |
| `MOVE A B` | `Reg(A):=Reg(B)` |
| `ADD A B C` | `Reg(A):=INT2FIX(FIX2INT(Reg(B))+FIX2INT(Reg(C)))` |
| `SUB A B C` | `Reg(A):=INT2FIX(FIX2INT(Reg(B))-FIX2INT(Reg(C)))` |
| `MUL A B C` | `Reg(A):=INT2FIX(FIX2INT(Reg(B))*FIX2INT(Reg(C)))` |
| `DIV A B C` | `Reg(A):=INT2FIX(FIX2INT(Reg(B))/FIX2INT(Reg(C)))` |
| `MOD A B C` | `Reg(A):=INT2FIX(FIX2INT(Reg(B))%FIX2INT(Reg(C)))` |
| `LT A B C` | `Reg(A):=INT2FIX(FIX2INT(Reg(B))<FIX2INT(Reg(C)))` |
| `LE A B C` | `Reg(A):=INT2FIX(FIX2INT(Reg(B))<=FIX2INT(Reg(C)))` |
| `EQ A B C` | `Reg(A):=INT2FIX(FIX2INT(Reg(B))==FIX2INT(Reg(C)))` |
| `NOT A B` | `Reg(A):=INT2FIX(not(FIX2INT(Reg(B))))` |
| `UNM A B` | `Reg(A):=INT2FIX((-1)*FIX2INT(Reg(B)))` |
| `JMPEQ0 A B` | `if FIX2INT(Reg(A))==0 then pc=pc+B` |

Figure 6.1: Extended instructions of a byte-code interpreter

**Instructions** Here, we explain how each extended instruction in LL0 corresponds to the byte-codes. We assume that each variable $x$ is assigned to a Register `Reg(`$n$`)`, and this correspondence is managed by toReg such as toReg$(x) = n$. In addition, we use a function newReg() to obtain $i$ such that `Reg(`$i$`)` is not used. To translate expressions into byte-codes [7], we define the translation exprBytes($target$, $expr$) that takes a target register number $target$ and an expression $expr$, and returns a sequence of byte-codes as shown in Figure 6.2, where each sub-expression of the expression is rewritten before applying to the translation exprBytes as follows:

$$
\begin{array}{lcl}
e_1 \texttt{ > } e_2 & \longrightarrow & e_2 \texttt{ < } e_1 \\
e_1 \texttt{ >= } e_2 & \longrightarrow & e_2 \texttt{ =< } e_1 \\
e_1 \texttt{ != } e_2 & \longrightarrow & \texttt{not(}e_1 \texttt{ == } e_2\texttt{)} \\
e_1 \texttt{ or } e_2 & \longrightarrow & e_1 \texttt{ + } e_2 \\
e_1 \texttt{ and } e_2 & \longrightarrow & e_1 \texttt{ * } e_2 \\
\texttt{(int)}e_1 & \longrightarrow & e_1
\end{array}
$$

By using the translation exprBytes, we explain how each extended instruction corresponds to a byte-code sequence:

- $x[p] =$ `(int)`$e$

$$
\begin{array}{lcl}
\mathsf{exprBytes}(target, i) & \stackrel{\text{def}}{=} & \texttt{MKVAL}\ \ target\ \ i \quad \text{where } i \text{ is an attribute} \\
\mid \mathsf{exprBytes}(target, x) & \stackrel{\text{def}}{=} & \texttt{MOVE}\ \ target\ \ \mathsf{toReg}(x) \\
\mid \mathsf{exprBytes}(target, \texttt{L}[n]) & \stackrel{\text{def}}{=} & \texttt{MOVE}\ \ target\ \ n \\
\mid \mathsf{exprBytes}(target, \texttt{R}[n]) & \stackrel{\text{def}}{=} & \texttt{MOVE}\ \ target\ \ m \quad \text{where } m = \texttt{MAX\_PORT} + n \\
\mid \mathsf{exprBytes}(target, op\ e_1) & \stackrel{\text{def}}{=} & \mathsf{let} \\
\quad \text{where } op \in \{\texttt{-}, \texttt{not}\} & & \quad reg_1 = \mathsf{newReg}();\ code_1 = \mathsf{exprBytes}(reg_1, e_1); \\
& & \quad inst = \{(\texttt{-}, \texttt{UNM}),\ (\texttt{not}, \texttt{NOT})\}[op] \\
& & \mathsf{in} \\
& & \quad code_1\ \ inst\ \ target\ \ reg_1 \\
& & \mathsf{end} \\
\mid \mathsf{exprBytes}(target, e_1\ op\ e_2) & \stackrel{\text{def}}{=} & \mathsf{let} \\
\quad \text{where } op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, & & \quad reg_1 = \mathsf{newReg}();\ code_1 = \mathsf{exprBytes}(reg_1, e_1); \\
\quad \texttt{<}, \texttt{<=}, \texttt{==}\} & & \quad reg_2 = \mathsf{newReg}();\ code_2 = \mathsf{exprBytes}(reg_2, e_2); \\
& & \quad inst = \{(\texttt{+}, \texttt{ADD}),\ (\texttt{-}, \texttt{SUB}),\ (\texttt{*}, \texttt{MUL}),\ (\texttt{/}, \texttt{DIV}), \\
& & \qquad (\texttt{\%}, \texttt{MOD}),\ (\texttt{<}, \texttt{LT}),\ (\texttt{<=}, \texttt{LE}),\ (\texttt{==}, \texttt{EQ})\}[op] \\
& & \mathsf{in} \\
& & \quad code_1\ \ code_2\ \ inst\ \ target\ \ reg_1\ \ reg_2 \\
& & \mathsf{end}
\end{array}
$$

Figure 6.2: The translation exprBytes from expressions into byte-code sequences

Assume a register number $r = \mathsf{Reg}()$ and a byte-code sequence $code = \mathsf{exprBytes}(r, e)$. When $x$ is assigned to $\texttt{Reg}(i)$, then the instruction corresponds to the following byte-code sequence:

$code$ `MOVEP i p r`

- $v = \texttt{(int)}e$

  The same as the case of $x[p] = \texttt{(int)}e$ let a register number $r = \mathsf{Reg}()$ and a byte-code sequence $code = \mathsf{exprBytes}(r, e)$. When $v$ is assigned to $\texttt{Reg}(i)$, then the instruction corresponds to the following byte-code sequence:

  $code$ `MOVE i r`

The function `evalCode` to evaluate instructions is extended in the part of the `case`-

branch as follows:

```
case MKVAL:
  Reg[code[pc+1]]=INT2FIX(code[pc+2]);
  pc+=3;
  break;
case MOVE:
  Reg[code[pc+1]]=Reg[code[pc+2]];
  pc+=3;
  break;
case ADD:
  Reg[code[pc+1]]=INT2FIX(FIX2INT(code[pc+2])+FIX2INT(code[pc+3]));
  pc+=4;
  break;
  ⋮
case UNM:
  Reg[code[pc+1]]=INT2FIX(-1*FIX2INT(code[pc+2]));
  pc+=4;
  break;
case JMPEQ0:
  if (!Reg[code[pc+1]])
    pc+=FIX2INT(code[pc+2]);
  pc+=3;
  break;
```

**Rule procedures** The extension of rule procedures is the `if`-statement that is written as follows:

```
rule α β {
  if (Cond_1) {···}
  elif (Cond_2) {···}
    ⋮
  elif (Cond_n) {···}
  else {···}
}
```

Let $block_i$ be a byte-code sequence for the `if`-block placed at the right of $Cond_i$, and *block* be one at the right of the `else`. In addition, let the length of the sequence $block_1, block_2, \ldots, block_n$ be $l_1, l_2, \ldots, l_n$ respectively. Then, the extended rule procedure corresponds to the following byte-code sequence:

```
let
  reg_1 = newReg();  code_1 = exprBytes(reg_1, Cond_1);
    ⋮
  reg_n = newReg();  code_n = exprBytes(reg_n, Cond_n);
in
  code_1
  JMPEQ0 reg_1  l_1 + 1
  block_1
  RETURN
  code_2
  JMPEQ0 reg_2  l_2 + 1
  block_2
  RETURN
    ⋮
  code_n
  JMPEQ0 reg_n  l_n + 1
  block_n
  RETURN
  block
  RETURN
end
```

**Runtime functions** `eval` The run time function `eval` is changed by using the type `VALUE` as follows:

```
void eval() {
  VALUE a1, a2;
  while (popActive(&a1, &a2)) {
    if (RBASIC(a2)->id != ID_NAME) {
      if (RBASIC(a1)->id != ID_NAME) {
        int i;
        for (i=0; i<Arities[RBASIC(a1)->id]; i++)
```

```
            Reg[i]=RAGENT(a1)->port[i];
          for (i=0; i<Arities[RBASIC(a2)->id]; i++)
            Reg[MAX_PORT+i]=RAGENT(a2)->port[i];
          evalCode(Code[RBASIC(a1)->id][RBASIC(a2)->id]);
          freeAgent(a1); freeAgent(a2);
        } /* The below is operations for x=t */
            ⋮
        }
      } else {
        /* operations for t=y and x=y */
          ⋮
      }
    }
}
```

## 6.6 Summary

In this chapter we extended interaction nets so that they can be used as a programming language. The first extension was nested pattern matching that is conservative, thus those rules can be translated into rules in the original interaction nets. Next we introduced agents that optionally have attributes, which are values of base type: integers, and interaction rules with these attributes and conditions. Finally, we extended the execution model to evaluate those extensions.

# Chapter 7

# Results and future work

In this chapter we implement a multi-threaded interpreter for interaction nets that uses LL0 as bytecode. As a result we obtain an improved performance in both sequential and parallel execution in comparison to existent interaction net evaluators as well as fully fledged programming languages SML and Python. Finally, we introduce some possible optimisations and extensions to enhance performance.

## 7.1 Interpreter for interaction nets with LL0

In this section we introduce a byte-code interpreter for interaction nets by using LL0 as an intermediate language. We use the C language to obtain efficient computation, and flex and bison to parse the syntax of interaction nets.

### 7.1.1 Sequential execution model

The implementation method of the sequential interpreter is similar to the standardised implementation model introduced in Sections 5.4 and 6.5.3, save some extensions which we discuss below. First, we introduce a virtual machine for executing our bytecodes while we look ahead to facilitate parallel execution where bytecodes can be evaluated by each thread locally.

**Data-structure** We represent agent nodes by using the same data-structure used in the standardised model where we fix the number of ports and we pre-populate the heap with these nodes. The fixed-size node representation has the disadvantage of using more space than needed, but the advantage of being able to manage and reuse nodes in a simpler way [35]. In comparison to Undirected encoding methods, Directed encoding methods introduce more name and indirection nodes. In order to separate these (name

| | in$^2$ | Single | Value | Single/Value |
|---|---|---|---|---|
| $F_{32}$ | 1.37 | 1.49 | 1.29 | 1.16 |
| $F_{33}$ | 2.29 | 2.49 | 2.12 | 1.17 |
| $F_{34}$ | 3.80 | 4.15 | 3.49 | 1.19 |
| $A(3,10)$ | 1.42 | 1.58 | 1.52 | 1.04 |
| $A(3,11)$ | 5.73 | 6.39 | 6.06 | 1.05 |
| $A(3,11)$ | 24.01 | 26.14 | 24.34 | 1.07 |
| `2 7 6 I I` | 0.71 | 1.28 | 1.18 | 1.08 |
| `2 7 7 I I` | 2.13 | 3.68 | 3.48 | 1.06 |

Table 7.1: The execution time in seconds on the standardised implementation model

and indirection) nodes from agent nodes, we prepare another heap and use the VALUE type introduced in Section 6.5. This separation simplifies the management of attributes and contributes to efficient computation.

Table 7.1 shows the execution time in minutes of running the benchmark programs in the standardised sequential implementation model using the VALUE type (labelled as VALUE). These were also compiled with gcc's `-O3` option. We see that this sequential implementation runs about from 4% to 19% faster and executes faster than in$^2$ in the case of the Fibonacci number.

**Virtual machine** A virtual machine (VM) manages the following components:

- A register `Reg`,
- LIFO stack for equations,
- Heaps for agents and names.

We represent our virtual machine using the code fragment:

```
typedef struct {
  Heap agentHeap, nameHeap; // heaps for agents and names
  Active *eqStack;          // equation stack
  VALUE Reg[REG_SIZE];      // Register
} VirtualMachine;
```

Figure 7.1 illustrates the virtual machine in contrast to the net configuration of the standardised implementation model (Figure 3.6).

Our virtual machine uses the following runtime functions:

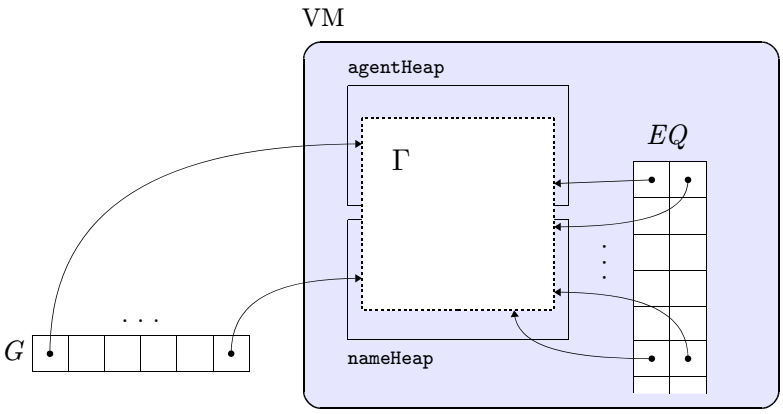
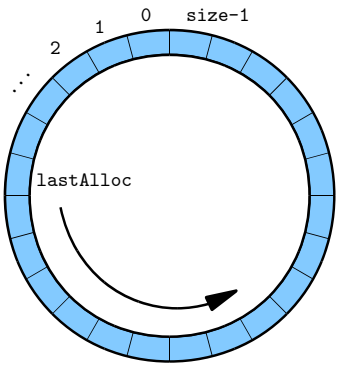



Figure 7.1: Virtual machine in the sequential execution model

```
VALUE mkAgent(VirtualMachine *vm, int id);
VALUE mkName(VirtualMachine *vm);
VALUE mkInd(VirtualMachine *vm);
void pushActive(VirtualMachine *vm, VALUE a1, VALUE a1);
int popActive(VirtualMachine *vm, VALUE *a1, VALUE *a1);
void evalCode(VirtualMachine *vm, int *code);
void eval(VirtualMachine *vm);
```

**Memory management** Taking account of parallel execution, we use a ring buffer for heaps:

```
typedef struct {
  VALUE *heap;
  int lastAlloc;
  unsigned int size;
} Heap;
```




In the `Heap` structure, `heap` is assigned to a large array of either `Agent` or `Name` that are cast by `(VALUE *)`, `lastAlloc` is the last used index, and `size` is the size of the heap.

To show that a node of the heap is available to use, we use the last bit of the `id` as a bit flag:

```
#define FLAG_AVAIL 0x01 << 31
#define IS_FLAG_AVAIL(a) ((a) & FLAG_AVAIL)
#define SET_FLAG_AVAIL(a) ((a) = ((a) | FLAG_AVAIL))
```

```
#define TOGGLE_FLAG_AVAIL(a) ((a) = ((a) ^ FLAG_AVAIL))
```

To allocate an agent node, we start to search an available node from `lastAlloc` and change the bit flag of the returned node to `0`. The `id` recovers the original meaning:

```
VALUE myallocAgent(Heap *hp) {
  int idx = hp->lastAlloc;
  int i;
  for (i=0; i < hp->size; i++) {
    if (IS_FLAG_AVAIL(((Agent *)hp->heap)[idx].basic.id)) {
      TOGGLE_FLAG_AVAIL(((Agent *)hp->heap)[idx].basic.id);
      hp->lastAlloc = idx;
      return (VALUE)&(((Agent *)hp->heap)[idx]);
    }
    idx++;
    if (idx >= hp->size)
      idx -= hp->size;
  }
  puts("Critical ERROR");
  exit(-1);
}
VALUE myallocName(Heap *hp) {
      ⋮
    if (IS_FLAG_AVAIL(((Name *)hp->heap)[idx].basic.id)) {
      TOGGLE_FLAG_AVAIL(((Name *)hp->heap)[idx].basic.id);
      hp->lastAlloc = idx;
      return (VALUE)&(((Name *)hp->heap)[idx]);
    }
      ⋮
}
```

To dispose of an agent node, we set the bit flag for availability into `1` again:

```
void myfree(VALUE ptr) {
  TOGGLE_FLAG_AVAIL(RBASIC(ptr)->id);
}
```

**Execution of bytecodes** In order to avoid inefficient switch-statements in our generated bytecodes, we use a method known as *threaded code* [11] which uses the goto construct to jump execution to some labelled block of code. In the C language, threaded code is written by replacing each case-branch with a label; a goto pointer to a label `label` is obtained by `&&label`. Following this technique, we write our runtime bytecode evaluation function `evalCode` as follows:

```
void evalCode(int *code) {
  static const void *table[] = {
    &&E_MKAGENT, &&E_MKNAME, &&E_MKIND, &&E_FREE, &&E_MOVEP,
    &&E_CHGID, &&E_PUSH, &&E_RETURN};
  int pc=0; // program counter
  goto *table[code[pc]];

 E_MKAGENT:
  Reg[code[pc+1]]=mkAgent(code[pc+2]);
  pc+=3; goto *table[code[pc]];
 E_MKNAME:
  Reg[code[pc+1]]=mkName();
  pc+=2; goto *table[code[pc]];
 E_MKIND:
  Reg[code[pc+1]]=mkInd();
  pc+=2; goto *table[code[pc]];
    ⋮
 E_RETURN:
  return;
}
```

### 7.1.2 Parallel execution model

In this section we discuss a multi-threaded parallel execution model of our bytecode interpreter for shared memory multiprocessors.

This model has the following objects:

- Multiple virtual machines,
- A thread pool for those virtual machines,

- Global equation stack *GlobalEQ*,
- Global array for the interface `G`.

We have multiple virtual machines that are managed in a thread pool. Each virtual machine has heaps which store graph elements and the net is constructed by connecting those elements. The global equation stack is used as a buffer to give and take equations among those virtual machines. Figure 7.2 illustrates a configuration where the thread pool has two virtual machines. Notice that a net may be distributed across the heaps of various VMs and an equation stack *EQ* of one VM may contain a pointer into the heap of another VM. In this figure, Γ represents the net and therefore it is composed of (initialised parts of) the heaps of the two virtual machines.

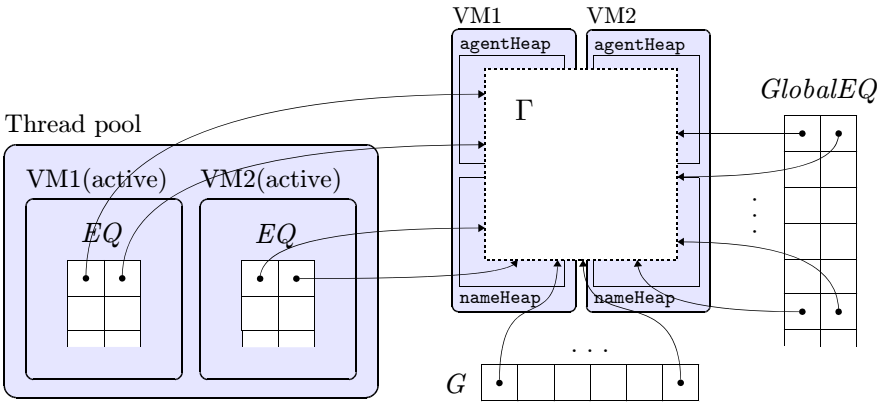



Figure 7.2: Configuration in the multi-threaded parallel execution model

Next, we explain the inter-relations between instance of virtual machines:

**State of threads:** A thread can be in either of two states: 1) *active* when its equation stack is not empty and 2) *sleep* when its stack and *GlobalEQ* is empty. A thread in sleep state switches to active only when it receives a signal `notEmpty`. Intuitively, the `notEmpty` signal is broadcast, when the *GlobalEQ* is not empty, to all VMs that have an empty local stack.

**Evaluation of equation in stacks:** Each virtual machine evaluates equations in its own stack. When the stack becomes empty, the VM tries to get an equation from *GlobalEQ*. When the *GlobalEQ* is also empty the state of the virtual machine switched to sleep and it remains in the sleep state until it receives a `notEmpty` signal. Figure 7.3 illustrates this behaviour.

**Stacking equations:** When equations are created by `evalCode` they are pushed onto

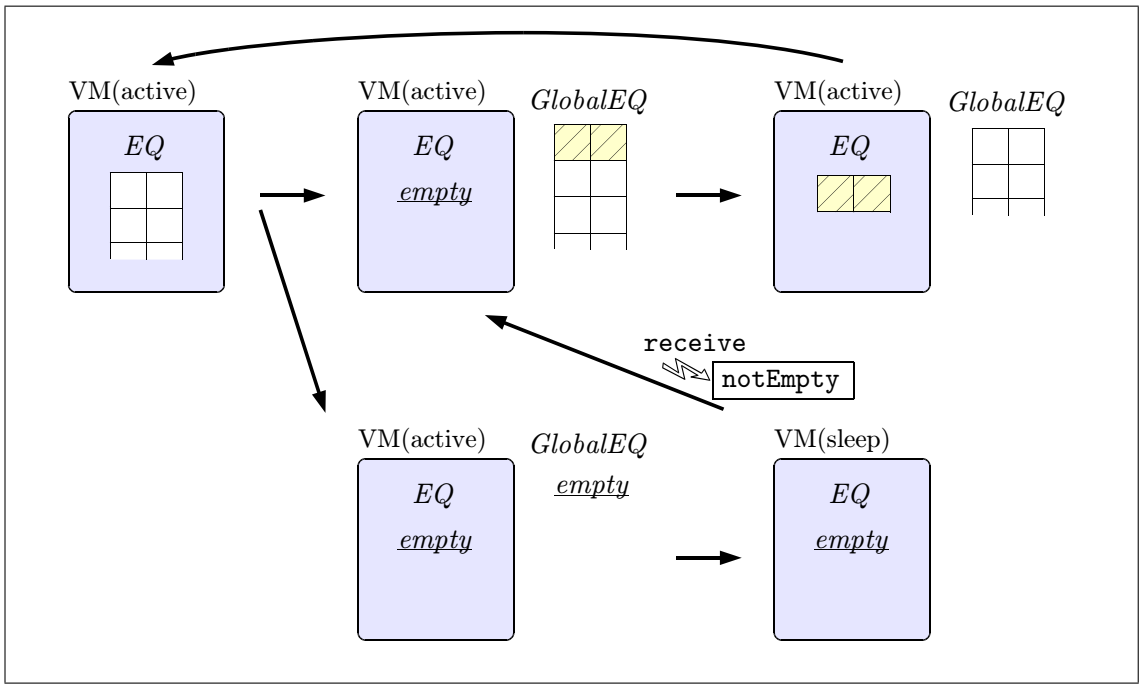



Figure 7.3: Transition of states and equation stacks

VM's local stack. If there exist some slept VMs, other equations can be pushed onto the *GlobalEQ* and this will trigger the broadcast of the `notEmpty` signal. In any case, at least one equation is pushed on the VM's local stack if one or more equations are created. Figure 7.4 illustrates this stacking mechanism.

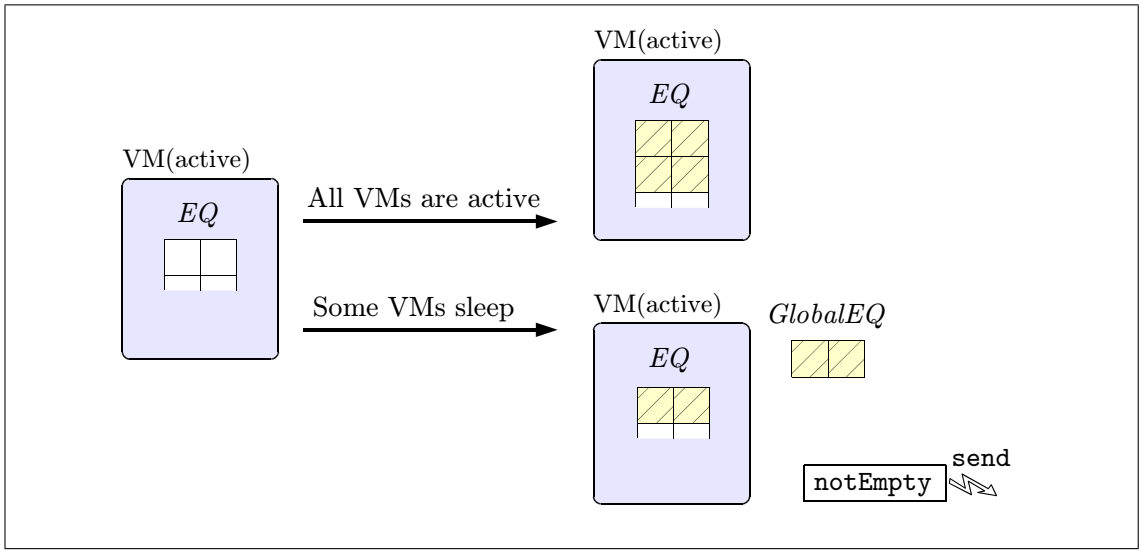



Figure 7.4: Stacking active pairs according to the condition of slept virtual machines

**Termination:** The evaluation finishes when all virtual machines sleep.

Communication between the global stack *GlobalEQ* and virtual machines requires syn-

chronisation in addition to the signalling. In the rest of this section we will discuss the synchronisation mechanism which we deploy:

**Transitions of B.1 and B.2** In parallel execution the operations of the transitions B.1 and B.2 given in Figure 4.2, which correspond to rules Var1 and Var2 in Simpler textual calculus (Definition 4.4.3), have critical sections since names can be shared by two active pairs. These are locked lightly by using an atomic operation like CAS, as shown in Section 4.5.3. The ports of agents and names can be changed by many threads simultaneously and thus the declaration of the `port` structures requires synchronisation code. We use `volatile` modifier as follows:

```
typedef struct {
  Basic basic;
  volatile VALUE port;
} Name;
typedef struct {
  Basic basic;
  volatile VALUE port[MAX_PORT];
} Agent;
```

**Global equation stack** *GlobalEQ* The Global equation stack *GlobalEQ* can be accessed by many threads and therefore it requires synchronisation so that it can be managed by only one thread at a time. This is realised by locks such as Mutex.

**Heaps in virtual machines** The allocation of nodes in a VM's heap is performed within the same VM and therefore synchronisation is not required. On the other hand, the disposing of those nodes is performed by other threads. Still, synchronisation is not required since those nodes are managed by ring buffers and the disposing does not affect the start index `lastAlloc` to check a ready-to-use node. When the value of the searching index `idx` which starts from `lastAlloc` is overlapped by the node index disposed of, the node will not be regarded as ready-to-use. This overlap can happen as many times as threads dispose of nodes during the searching of the ready-to-use nodes, however these opportunities are greatly less than the size of the heap. In order to obtain efficient computation, therefore, we do not use any lock mechanism, and thus use *wait-free* algorithm [53], assuming that each virtual machine has sufficiently large size heaps.

| | INET | amLight | Inpla | $\text{Inpla}_1$ | $\text{Inpla}_2$ | $\text{Inpla}_3$ | $\text{Inpla}_4$ | $\text{Inpla}_5$ |
|---|---|---|---|---|---|---|---|---|
| $F_{30}$ | 3.70 | 2.46 | 2.45 | 2.61 | 2.12 | 1.99 | 1.93 | 1.93 |
| $F_{31}$ | -[1] | 4.03 | 3.02 | 3.25 | 2.46 | 2.23 | 2.12 | 2.14 |
| $F_{32}$ | -[2] | 6.65 | 3.95 | 4.32 | 3.03 | 2.65 | 2.49 | 2.50 |
| $A(3,9)$ | 4.12 | 2.85 | 1.21 | 1.27 | 0.67 | 0.46 | 0.37 | 0.35 |
| $A(3,10)$ | 18.26 | 11.40 | 4.88 | 5.06 | 2.64 | 1.81 | 1.41 | 1.34 |
| $A(3,11)$ | 66.79 | 46.30 | 19.19 | 20.65 | 10.82 | 7.45 | 5.92 | 5.72 |
| `2 6 7 I I` | 1.60 | 1.49 | 1.51 | 1.57 | 1.27 | 1.23 | 1.33 | 1.23 |
| `2 7 6 I I` | -[2] | 4.01 | 2.61 | 2.69 | 1.80 | 1.88 | 1.64 | 1.81 |
| `2 7 7 I I` | -[2] | 11.96 | 6.07 | 6.19 | 3.77 | 2.88 | 2.90 | 2.80 |

[1] Segmentation fault (core dumped)

[2] Heap exceeds limit: <8388608>

Table 7.2: The execution time in seconds on interaction nets evaluators

### 7.1.3 Experimental results

We implemented a multi-threaded parallel interpreter of the bytecode, called *Inpla*, with gcc 4.6.3 and the Posix-thread library. In this section, we compare the execution time of Inpla with other evaluators and interpreters. First, in executions of the pure interaction nets, we take INET and amineLight and compare Inpla with those by using the benchmark programs also used in Section 4.5.3 – Fibonacci number, Ackermann function and application of Church numerals. Next, we compare Inpla with Standard ML of New Jersey (SML) [54] and Python [61] in the extended framework of interaction nets given in Chapter 6 which includes integer numbers and lists. SML is a functional programming language and it has the eager evaluation strategy that is similar to the execution method in interaction nets. Python is a widely-used interpreter, and thus the comparison with Python gives a good indication on efficiency. Here we benchmark the Fibonacci number, the Ackermann function on integer numbers, the Bubble Sort algorithm and the Quick Sort algorithm.

The benchmark programs were run on a Linux PC (2.4GHz, Core i7, 16GB) and the execution time was measured using the UNIX `time` command. The version of Python is 2.7.3, and SML is v110.74.

**Pure interaction nets** Table 7.2 shows execution time in seconds among interaction nets evaluators: INET, amineLight and Inpla. In the table the subscript of Inpla gives the number of threads in the thread pool, for instance $\text{Inpla}_3$ means that it was executed

by using three threads. The '-' means no execution time due to some error.

We see that Inpla runs faster than INET since Inpla is a refined version of amineLight, which is the fastest interaction nets evaluator [30].

In comparison to amineLight, Inpla compiles nets into bytecodes, whereas amineLight interprets those directly. We see that in the case of `2 6 7 I I`, amineLight executes faster than Inpla; this is because Inpla compiles the program and this compilation contributes towards obtaining efficient computation. For the same reason, the execution of $F_{30}$ and `2 6 7 I I` gives almost the same execution time. However, there is a significant increase in the performance of Inpla when we increase the size of computation. For example, Inpla runs 2.4 times faster than amineLight to execute $A(3, 11)$.

Next, we discuss the parallel execution in Inpla. Figure 7.5 shows the average of speedup as the ratio $S(n) = \dfrac{T(0)}{T(n)}$ where $T(0)$ is the best execution time in sequential and $T(i)$ is an execution time by $i$-threads. Generally, since Core i7 processor has four cores, it tends to reach the peak in the four threads execution. In the case of Ackermann function, those speedups have the best trends that are close to $n$-fold increasing to $n$-threads executions. In the other cases, while the increasing trend is calm, the execution becomes faster according to the number of threads in the pool. In contrast to the execution of Ackermann function, these executions require quite a huge number of nodes in the heaps. Actually the computations of Fibonacci number require 100,000,000 nodes as the amount of the agent and the name heaps in each virtual machine, and the computations of Applications of Church numerals require 60,000,000 nodes, whereas those of Ackermann function are performed within 100,000 nodes, which is a normal setting in Inpla. Those could induce a low hit of the cache memory and the speedup ratio would be calm.

**Computation on integer numbers and lists** Here we compare Inpla with SML and Python. Table 7.3 shows execution time in seconds for SML, Python and Inpla in computation on integer numbers and lists. Bubble Sort BS $n$ and Quick Sort QS $n$ sort lists that have $n$-elements which are randomly generated. These programs use integer numbers, whereas programs in Table 7.2 use unary natural numbers. In interaction nets these are managed by using attributes discussed in Chapter 6.

First, we examine the computation results in the sequential execution. As shown in the table, SML computes those arithmetic functions fastest. SML computes Fibonacci number and Ackermann function around 19 times and 4.3 times faster at best than Inpla, respectively. In the computation in Inpla, the functions and integer numbers are represented by agents, and those agents are consumed and re-produced repeatedly dur-

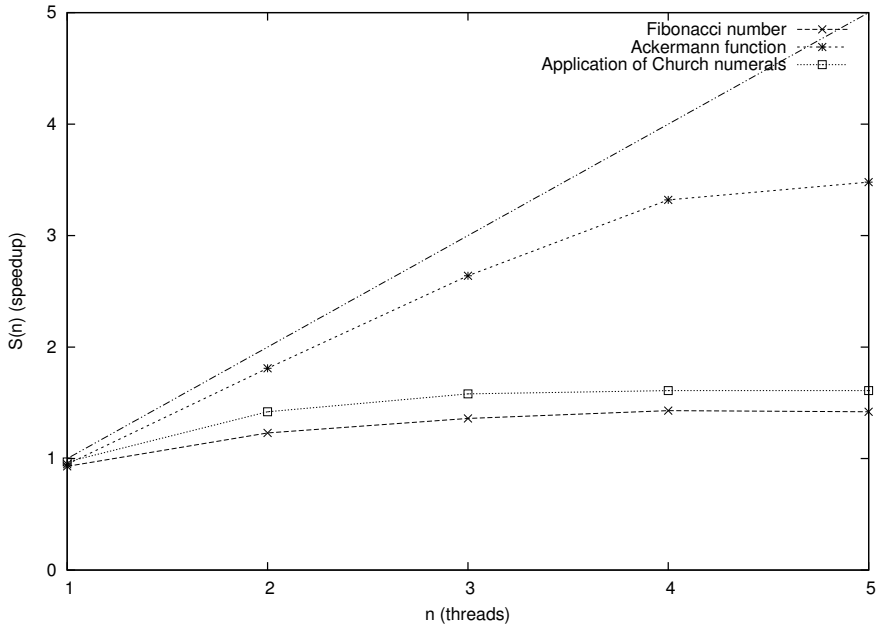



Figure 7.5: The speedup in the multi-threads executions

ing computation. Thus the execution time becomes slower eventually, compared to the execution in SML that performs computation by function calls and managing stacked arguments. This could be improved by re-using agents as discussed in Section 7.2.1. In comparison with Python, Inpla computes those functions from about 17% faster at best to about 6% faster at worst. This shows that Inpla can indeed be used in practical and not only for theoretical investigation

The two sort algorithms are special cases in that interaction nets are efficient to implement the algorithms. We see that in the case of Bubble Sort, Inpla performs a little faster than SML and 3.3 times faster than Python. In the case of Quick Sort, Inpla performs around 31 times faster than Python. Although it performs around 2.6 times slower than SML, this is improved in parallel execution: Inpla performs around 3.47 times faster at best to around 1.04 times faster at worst.

Next we examine the parallel execution of Inpla. Table 7.6 shows the speedup in the multi-threads executions. In the case of Fibonacci number, Bubble Sort and Quick Sort, the speedup are also close to the $n$-fold increasing, which is the best performance in parallel execution. Thus we expect parallel execution to obtain efficient computation.

On the other hand, the computations of Ackermann function becomes less efficient with more number of threads execution. Figure 7.7 shows the trends on execution of those benchmark programs in sequential and parallel where we assume unbounded resources in terms of the number of processing elements available. In Ackermann function, there

| | SML | Python | Inpla | $\text{Inpla}_1$ | $\text{Inpla}_2$ | $\text{Inpla}_3$ | $\text{Inpla}_4$ | $\text{Inpla}_5$ |
|---|---|---|---|---|---|---|---|---|
| $F_{37}$ | 0.43 | 8.26 | 7.05 | 8.10 | 4.20 | 3.05 | 2.36 | 2.43 |
| $F_{38}$ | 0.67 | 13.33 | 11.45 | 13.12 | 6.81 | 4.73 | 3.67 | 3.50 |
| $F_{39}$ | 1.08 | 21.87 | 20.49 | 21.90 | 10.73 | 7.49 | 5.84 | 5.85 |
| $A(3,6)$ | 0.05 | 0.05 | 0.03 | 0.03 | 0.07 | 0.25 | 0.23 | 0.27 |
| $A(3,7)$ | 0.05 | -[1] | 0.07 | 0.08 | 0.18 | 0.73 | 0.72 | 1.16 |
| $A(3,8)$ | 0.06 | -[1] | 0.26 | 0.28 | 0.63 | 1.90 | 3.06 | 3.63 |
| BS 20000 | 8.60 | 28.06 | 8.45 | 9.43 | 4.89 | 3.37 | 2.59 | 2.57 |
| BS 30000 | 21.43 | 63.28 | 19.01 | 21.20 | 10.97 | 7.55 | 6.16 | 6.48 |
| QS 400000 | 0.65 | 49.65 | 1.54 | 1.62 | 0.88 | 0.70 | 0.62 | 0.76 |
| QS 500000 | 0.91 | 77.56 | 2.53 | 2.68 | 1.25 | 0.99 | 0.79 | 0.94 |

[1] RuntimeError: maximum recursion depth exceeded

Table 7.3: The execution time in seconds on interpreters

is no significant difference in sequential and parallel execution. This is one reason why the parallel execution does not have good performance. In addition, this is caused by the realisation of the computation for $A(m-1, A(m, n-1))$, which is a one step part of Ackermann function, represented as the following two equations as shown in Example 6.3.7:

$$\texttt{A}(w, r) = \texttt{(}m'\texttt{)},\ \texttt{A((}n'\texttt{)}, w) = \texttt{(}m\texttt{)}\ \texttt{where}\ m' = m\texttt{-1}\ \ n' = n\texttt{-1}$$

The first equation $\texttt{A}(w, r) = \texttt{(}m'\texttt{)}$ takes the computation result of the second equation, which is corresponding to $A(m, n-1)$, via the name $w$. Actually the first equation is reduced to $\texttt{A2((}m'\texttt{)}, r) = w$, and then it waits the $w$. As for the second equation $\texttt{A((}n'\texttt{)}, w) = \texttt{(}m\texttt{)}$, it reaches to the step of $A(m-1, A(m, (n-1)-1))$ again unless $m$ or $n-1$ is 0, and thus the two equations are produced that one waits the computation result of the other. In the implementation of Inpla, this means that, when two equations are produced by an active thread, then the thread would sleep while waking up another slept thread. After that, the waked-up thread would produce the two equations again, and it would sleep while waking up another thread. This is repeated until all of the computation are finished, causing the overhead. This vicious repeat could not occur, when a large scale computation is performed at once, because each thread could have huge numbers of active pairs and there is not so many opportunities of the sleep.

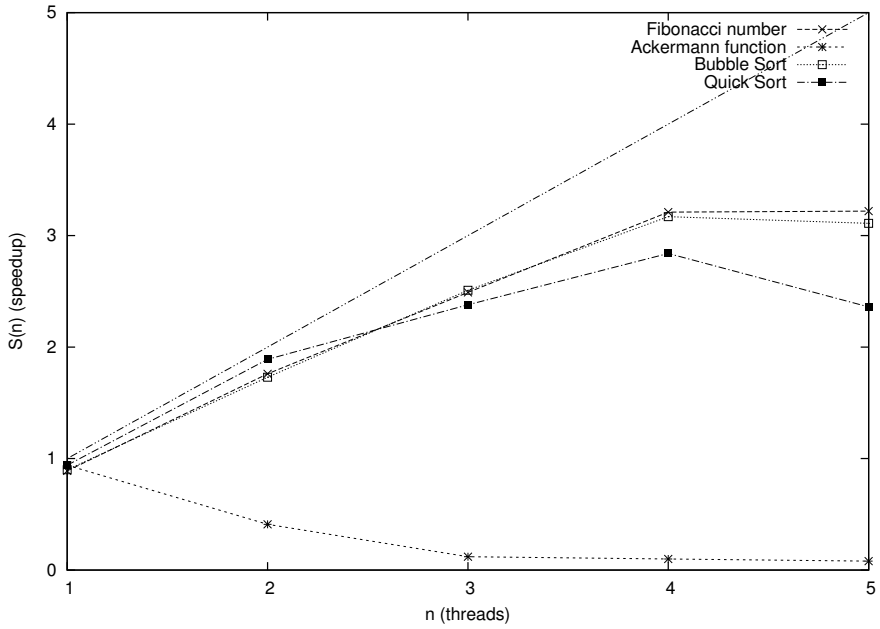



Figure 7.6: The speedup in the multi-threads executions using attributes

## 7.2 Future work

In this section we introduce other possible optimisations and extensions in terms of efficient computation that we leave for future work.

### 7.2.1 Reuse optimisation

Once a net is compiled into an instruction list of LL0, operations such as producing, disposing and connecting ports of agents is done at the level of execution of those instructions.

Here, we take an interaction rule between `Add` and `S` as an example:

- $\texttt{Add}(x_1, x_2) = \texttt{S}(y) \Rightarrow x_2 = \texttt{S}(w), \texttt{Add}(x_1, w) = y.$

This compilation is illustrated in Example 5.2.5, and it is obtained as shown in Figure 7.8 (a). In the RHS of this rule, the same agents to the active pair occur. Thus, instead of producing new those agents, it is possible to reuse active pair agents as the new ones. Figure 7.8 (b) shows the rewritten lists by reusing the active pair agents. The number of instructions decreases, and thus faster execution is expected.

In our language, moreover, an id of an agent node `a` is referred to `a[0]`, and thanks to their fixed arity number it is also possible to reuse an agent as another one. Therefore this method works as optimisation for rule procedures.

In the standardised implementation model, the index of the equation stack is managed by functions `pushActive` and `popActive`, and the instruction `stackFree()` is ignored.

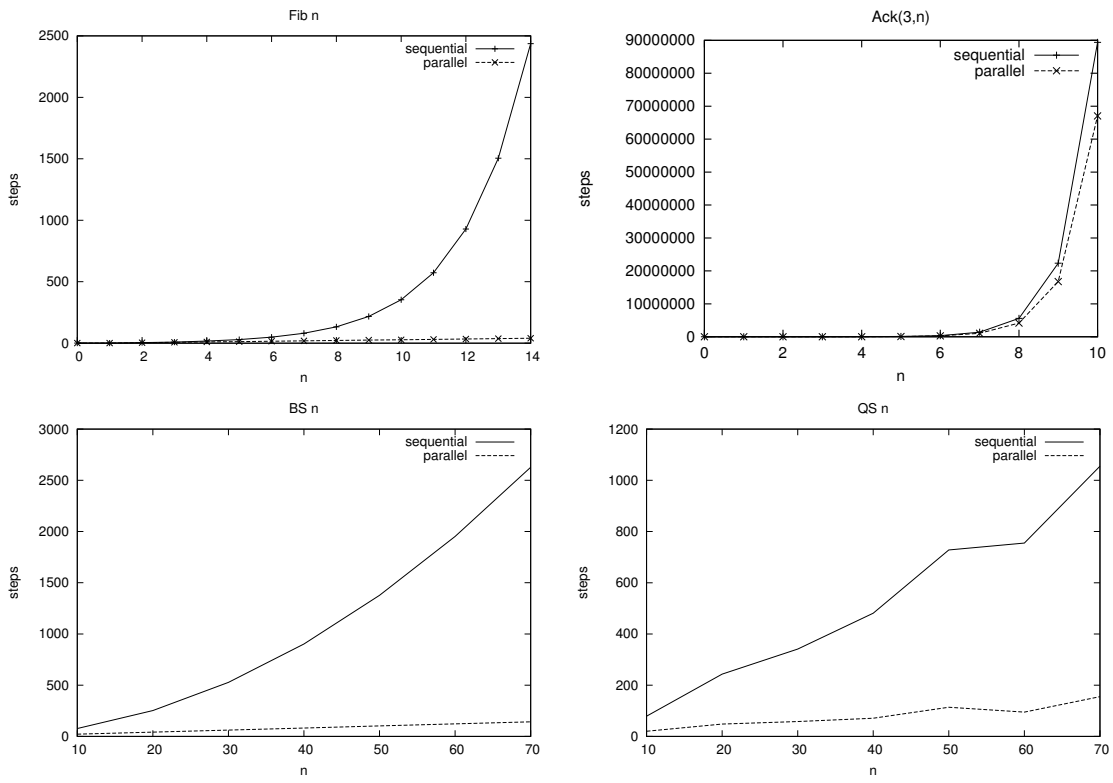



Figure 7.7: Execution steps on benchmark programs in sequential and parallel

(a) Instruction list

```
rule Add S {
  stackFree()
  w=mkName()
  aS=mkAgent(S)
  aS[1]=w
  push(L[2],aS)
  aAdd=mkAgent(Add)
  aAdd[1]=L[1]
  aAdd[2]=w
  push(aAdd,R[1])
  free(L)
  free(R)
}
```

(b) Optimised one by the reuse method

```
rule Add S {
  w=mkName()
  x2=StackL[2]
  tmpR=StackR
  StackR=tmpR[1]
  tmpR[1]=w
  push(x2,tmpR)
}
```

Figure 7.8: Rule procedures for the rule between `Add` and `S`

Here, we modify this implementation model to manage the top element of the stack explicitly.

**Explicitly managed equation stack** To make explicit operations for the stack, the following are introduced:

```
#define stackFree() Ptr_APS--
#define StackL ActivePairs[Ptr_APS].a1
#define StackR ActivePairs[Ptr_APS].a2
```

The macro `stackFree()` reduces the index of the equation stack. The `StackL` and `StackR` are replaced with elements on the current top of the stack. The pop stack function is not required since the elements in the top of the stack are referred by `StackL` and `StackR`.

In addition, each function for interaction rules does not require arguments `Agent *a1` and `Agent *a2` since those can be referred by `StackL` and `StackR`. The runtime function `eval` is also changed as follows:

```
void eval() {
  while (Ptr_APS >= 0) {
    if (StackR->id != ID_NAME) {
      if (StackL->id != ID_NAME) {
        R[StackL->id][StackR->id]();
      } else if (StackL->id == ID_INDIRECTION) {
        /* C.1 */
        Agent *tmpL = StackL;
        StackL = StackL->port[0];
        freeAgent(tmpL);
      } else {
        /* B.1 */
        StackL->port[0] = StackR;
        StackL->id = ID_INDIRECTION;
        stackFree();
      }
    } else if (StackR->id == ID_INDIRECTION) {
      /* C.2 */
      Agent *tmpR = StackR;
      StackR = StackR->port[0];
```

| | Single | Reuse | Single/Reuse |
|---|---|---|---|
| $F_{32}$ | 1.49 | 0.80 | 1.86 |
| $F_{33}$ | 2.49 | 1.31 | 1.90 |
| $F_{34}$ | 4.15 | 2.14 | 1.94 |
| $A(3,10)$ | 1.58 | 1.24 | 1.27 |
| $A(3,11)$ | 6.39 | 4.97 | 1.29 |
| $A(3,11)$ | 26.14 | 21.21 | 1.23 |
| `2 7 6 I I` | 1.28 | 1.23 | 1.04 |
| `2 7 7 I I` | 3.68 | 3.63 | 1.01 |

Table 7.4: The execution time in seconds on the single encoding method and the reuse method

```
        freeAgent(tmpR);
      } else {
        /* B.2 */
        StackR->port[0] = StackL;
        StackR->id = ID_INDIRECTION;
        stackFree();
      }
    }
  }
```

**Experimental results** Table 7.4 shows execution time in seconds of programs that are written manually in order to apply this method. Those are compiled with the `-O3` option.

The speedup in the case of Fibonacci number, Ackermann function and Application of Church numerals are about 1.9 times, 1.27 times and 1.02 times respectively. Although there is fluctuation in the effect, this method can improve the computation efficiency. Automatically applying this reuse optimisation is a future work.

### 7.2.2 Parallelism

In this section, we look where there is parallelism in interaction nets.

We introduced two rule sets for addition: one is in Figure 2.1 and alternative one is in Figure 2.3. First, we take the alternative one that is regarded as a sequential version of addition because the rules do not produce any active pairs. Actually, as shown in Figure 7.9, the computation of `add`$(m, n)$ has no scope for parallelism.

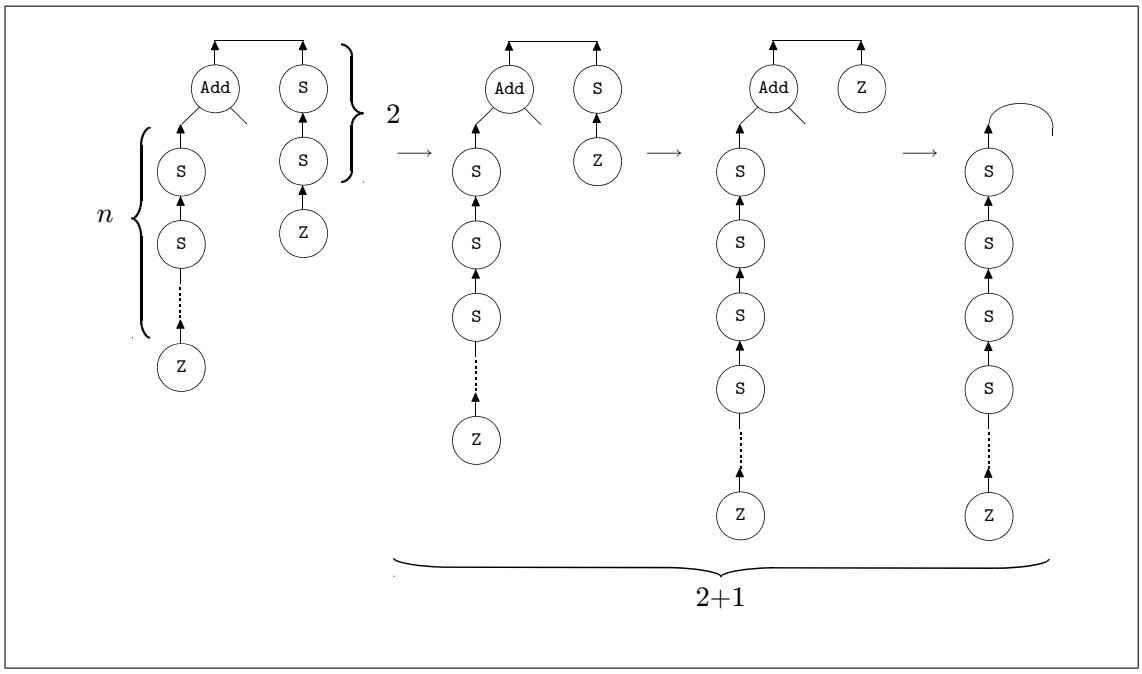



Figure 7.9: $\texttt{add}(\bar{2}, \bar{n})$ in a sequential version of addition

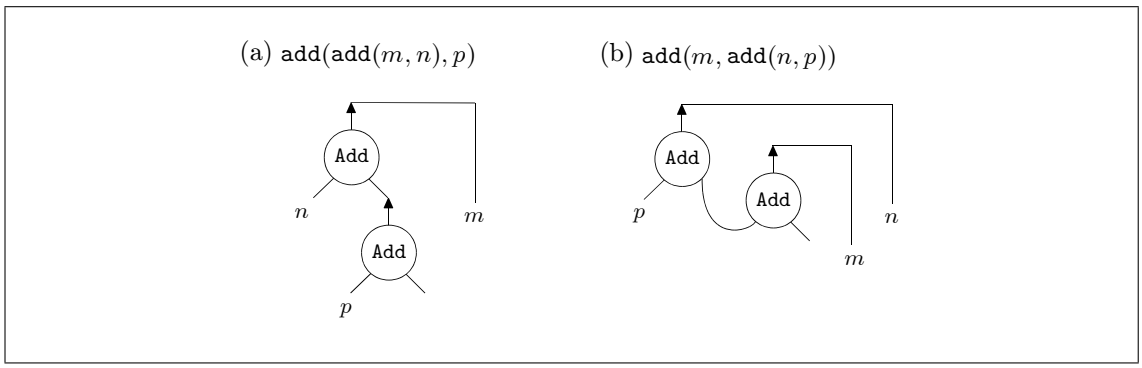



Figure 7.10: $\texttt{add}(\texttt{add}(m, n), p)$ and $\texttt{add}(m, \texttt{add}(n, p))$ in the alternative rules

Next we consider the cost of rewritings as the number of interactions. The net corresponding to $\texttt{add}(m, n)$ requires $m + 1$ interactions (Figure 7.9) regardless of the size of the net $\bar{n}$. Thus, the net corresponding to $\texttt{add}(\texttt{add}(m, n), p)$ in Figure 7.10 (a) requires $(m+1)+(m+n+1) = 2m+n+2$ interactions. This net is still sequential, and in parallel execution the cost is the same (thus $2m + n + 2$). If we use, however, the associative property of addition, now the net is corresponding to $\texttt{add}(m, \texttt{add}(n, p))$ in Figure 7.10 (b), then the situation changes significantly. The cost becomes $(m+1)+(n+1) = m+n+2$. Moreover, in parallel execution, it is $\texttt{max}(m+1, n+1)$. By applying the associative property of addition, not only it becomes more efficient sequentially, but also it becomes able to benefit from parallel evaluation.

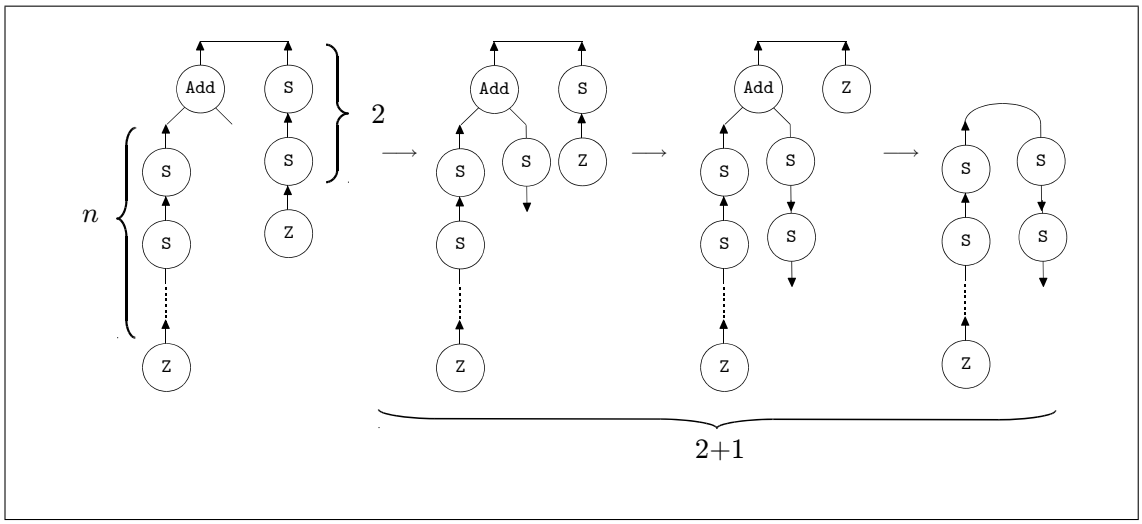



Figure 7.11: $\texttt{add}(\bar{2}, \bar{n})$ in a parallel version of addition

On the other hand, in the rule set in Figure 2.1, the net corresponding to $\texttt{add}(m, n)$ also requires $m+1$ interactions (Figure 7.11), however we call the set a parallel version of addition because it is possible to produce an active pair when the free port is connected to a principal port of an agent. For instance, the net corresponding to $\texttt{add}(\texttt{add}(m, n), p)$ requires $(m+1)+(m+n+1) = 2m+n+2$ interactions the same as the alternative rule set, however in parallel execution, as shown in Figure 7.12, the cost is $(m+1)+(n+1) = m+n+2$. In the case of $\texttt{add}(m, \texttt{add}(n, p))$, the cost is the same as the alternative version because the produced `S` by the rule is placed to the auxiliary port of another `Add` agent and it does not contribute to parallel execution.

Next we consider how those two versions of addition work in Fibonacci number. Figure 7.13 shows the cost that is required to obtain the calculation result in sequential and parallel versions of addition. In this graph, we assume unbounded resources in terms of the number of processing elements available. The cost in the parallel version is significantly less than the sequential version.

On the other hand, Table 7.5 shows the execution time in seconds by Inpla. In both cases the execution becomes faster by using several threads, however the parallel version is slower than the sequential one. This is because there are more active pairs for the parallel execution. Moreover produced active pairs by the rules have the same vicious repeat problem mentioned in Section 7.1.3.

We summarise this topic:

- some nets can use properties of the system (in this case associativity of addition) to get better sequential and parallel behaviours;

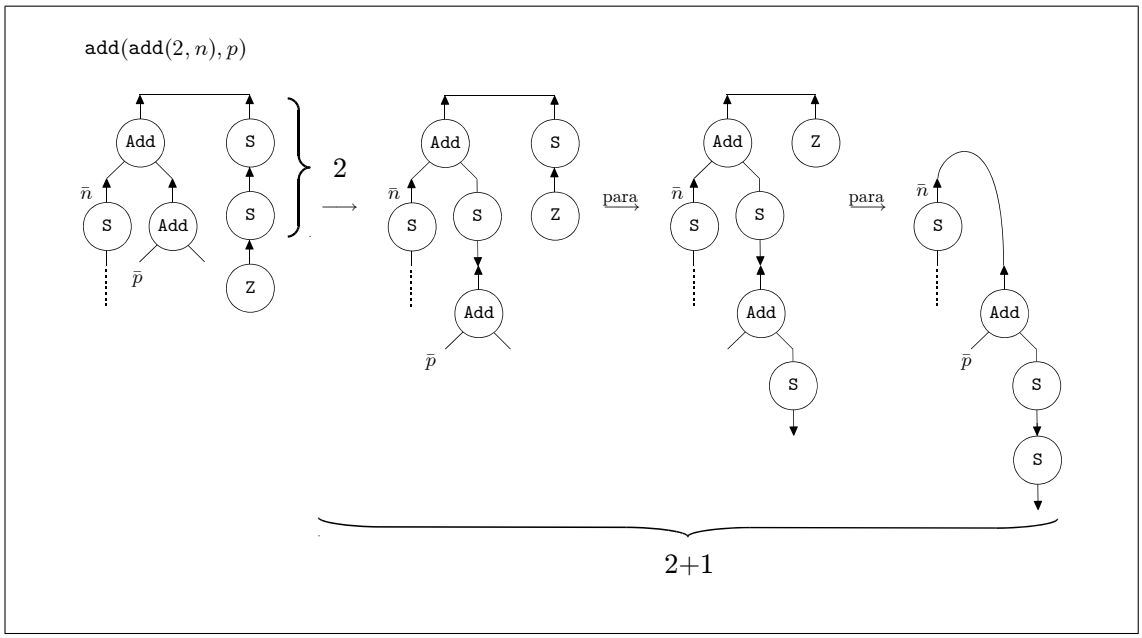



Figure 7.12: Parallel execution of $\texttt{add}(\texttt{add}(2, \bar{n}), \bar{p})$

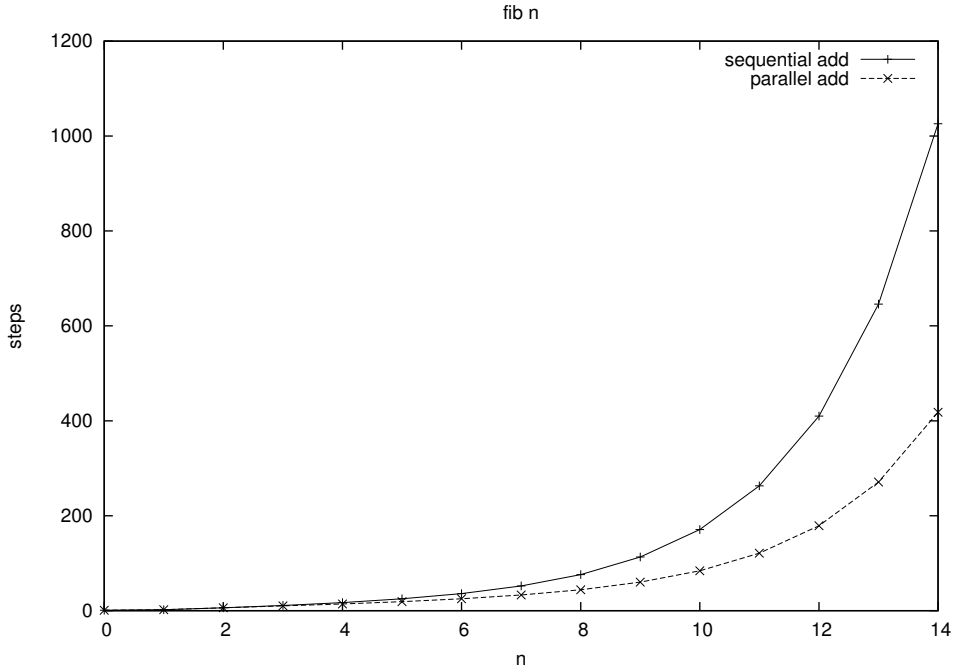



Figure 7.13: Behaviour of sequential and parallel versions of addition on Fibonacci function

- some systems can have modified rules that are more efficient, and also more appropriate to exploit parallelism.

Thus, we can choose rules to get better sequential and parallel behaviour, however the criteria which we should choose is one of future work.

| $F_{30}$ | $\text{Inpla}_1$ | $\text{Inpla}_2$ | $\text{Inpla}_3$ | $\text{Inpla}_4$ | $\text{Inpla}_5$ |
|---|---|---|---|---|---|
| Sequential ver. | 2.61 | 2.12 | 1.99 | 1.93 | 1.93 |
| Parallel ver. | 2.87 | 2.29 | 2.12 | 2.05 | 2.06 |

| $F_{31}$ | $\text{Inpla}_1$ | $\text{Inpla}_2$ | $\text{Inpla}_3$ | $\text{Inpla}_4$ | $\text{Inpla}_5$ |
|---|---|---|---|---|---|
| Sequential ver. | 3.25 | 2.46 | 2.23 | 2.12 | 2.14 |
| Parallel ver. | 3.70 | 2.73 | 2.45 | 2.34 | 2.34 |

| $F_{32}$ | $\text{Inpla}_1$ | $\text{Inpla}_2$ | $\text{Inpla}_3$ | $\text{Inpla}_4$ | $\text{Inpla}_5$ |
|---|---|---|---|---|---|
| Sequential ver. | 4.32 | 3.03 | 2.65 | 2.49 | 2.50 |
| Parallel ver. | 5.06 | 3.47 | 2.99 | 2.81 | 2.82 |

Table 7.5: Execution time in the multi-thread execution

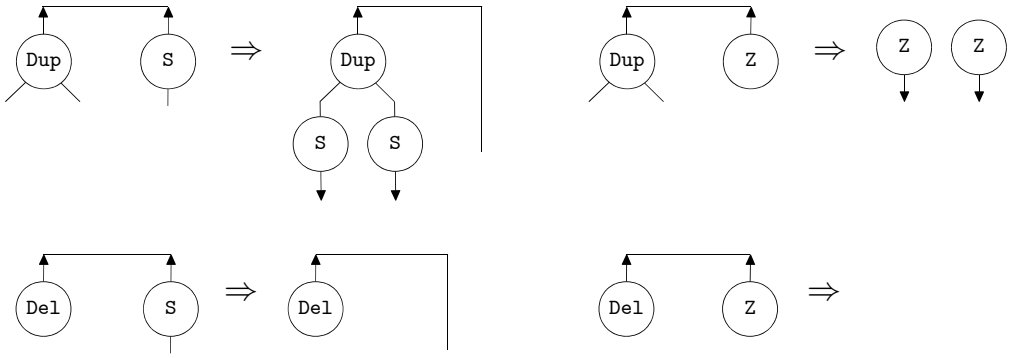
### 7.2.3 Algebraic datatypes and sharing

In this section, as another possible optimisation, we discuss sharing given nets defined by algebraic datatypes.

An algebraic datatype with $n$ constructors $C_1, \ldots, C_n$ can be represented by using $n$ agents $A_i$ where each agent $A_i$ has the same arity of $C_i$. For instance, unary natural numbers are defined as the following algebraic datatype `Nat`:

```
datatype Nat = Z | S of Nat;
```

and those are represented nets by using the agent `Z` and `S`. Those net are constituted only by connection between auxiliary ports and principal ports, thus those have no names. Therefore, it is possible to share pointers to those nets safely if we do not erase any of those contents. The following are rules for duplication and erasing:




Functions for those rules could be written by the pointer sharing:

```
void Dup_S(Agent *a1, Agent *a2) {
  pushActive(a1->port[0],a2);
```

```
  pushActive(a1->port[1],a2);
  freeAgent(a1);
}
void Dup_Z(Agent *a1, Agent *a2) {
  pushActive(a1->port[0],a2);
  pushActive(a1->port[1],a2);
  freeAgent(a1);
}
void Del_S(Agent *a1, Agent *a2) {
  // no operation
}
void Del_Z(Agent *a1, Agent *a2) {
  // no operation
}
```

Garbage nodes, thus unerased and unnecessary agents, are managed by a garbage collector. The garbage can be found with Mark-and-sweep method by recursively tree-walking on the interface (and active pair stacks if those are not empty). By using this method, the computation could be improved efficiently.

## 7.3 Summary

In this chapter, we implemented a multi-threaded interpreter of interaction nets that uses LL0 as bytecode, and we showed how our method improved the performance in sequential and parallel execution. We also introduced some possible optimisations and extensions in terms of efficient computation.

With respect to another improvement with LL0, we note the correspondence of LL0 to the standardised implementation written in the C language as mentioned in Sections 5.3 and 6.5.2. PIN [28], which is a bytecode interpreter, led to INET [29], which is a compiler to the C language. In the same way, a new compiler for interaction nets based on the LL0 language will be developed, followed by our interpreter Inpla.

# Chapter 8

# Conclusion

Interaction nets have been expected to give a new, alternative, theoretical foundation of sequential and parallel computation since they were proposed in 1990, particularly with respect to efficiency. To demonstrate this ideal, it is important to show that ideas work efficiently not only in theory but also in practice. This thesis has contributed to this research effort by providing more effective and simpler methods in the development of implementations of interaction nets. Our main contributions can be summarised as:

- We introduced a standardised implementation model. There are so many implementations of interaction nets, and they cannot be compared or analysed in a uniform way. Having a standard model for evaluation allows us to start to develop tools and techniques to reason about implementations. Of course, there are alternative implementation models waiting to be developed. However, we see this work as an important step to push forward the idea of a standard model—even if other models are developed later, the techniques provide an important start to this work.

- By using this model, we examined a number of interaction net evaluators that have been developed to date, and have demonstrated the necessity of our new method.

- In terms of sequential evaluation, our method is not necessarily the most efficient—however, it is simpler (in some cases significantly) than extant evaluators. Having a simple—perhaps the simplest—model allows us to see the essential structure of interaction, and moreover it is possible to perform evaluation in parallel naturally. The motivation to give a simple model is analogous to something like the Krivine machine for the $\lambda$-calculus for example (this machine is not the most efficient way to implement the $\lambda$-calculus, but it is important in the understanding of $\beta$-reduction).

- We introduced a new textual calculus that mirrors the implementation method. This is useful to investigate properties of an implementation from a theoretical perspective—it provides an interface between the theory and practice.

- We also introduced a bytecode execution model, which is called LL0. This offers not only efficient implementation, but also parallel implementation.

- We implemented a parallel evaluator, called Inpla. This is the fastest evaluator for interaction nets known to date. In comparison with Python, it is also faster. In comparison with SML, it tends not to be faster, but it can be faster if we use a specific, efficient, encoding of the algorithm.

- Finally, in the future works we gave significant evidence that using this model allows us to reason about and develop optimisations, such as the reuse optimisation of memory cells. This is another advantage to use LL0.

We hope that the methods proposed in this thesis could help push forward the development of interaction based evaluators, and inspire new work on parallel implementations of interaction nets and as a consequence parallel implementations of other programming languages through translation.

# Bibliography

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993. 74

[2] Vladimir Alexiev. *Non-deterministic interaction nets.* PhD thesis, University of Alberta, 1999. Adviser-Jia You. 7, 148

[3] José Bacelar Almeida, Jorge Sousa Pinto, and Miguel Vilaça. A tool for programming with interaction nets. *Electr. Notes Theor. Comput. Sci.*, 219:83–96, 2008. 4, 30

[4] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. Gödel's system T revisited. *Theor. Comput. Sci.*, 411(11-13):1484–1500, 2010. 23

[5] Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. PORGY: strategy-driven interactive transformation of graphs. In Rachid Echahed, editor, *Proceedings 6th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2011, Saarbrücken, Germany, 2nd April 2011.*, volume 48 of *EPTCS*, pages 54–68, 2011. 6

[6] Minero Aoki. Ruby source code: A full description. `http://i.loveruby.net/ja/rhg/book/`, 2004. Accessed: 12 August 2014. English translation: The Ruby Hacker's Guide [12]. 170, 171

[7] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques.* Cambridge University Press, New York, NY, USA, 1997. 174

[8] Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1998. 4, 5

[9] Alan Bawden. Connection graphs. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 258–265, 1986. 3

[10] Denis Bechet. Partial evaluation of interaction nets. In M. Billaud, P. Castéran, M. M. Corsini, K. Musumbu, and A. Rauzyand, editors, *Proceedings of the Second Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 331–338, 1992. 149

[11] James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973. 183

[12] Clifford Escobar Caoile, Robert Gravina, Vincent Isambart, and C.E. Thronton. The ruby hacker's guide. http://edwinmeyer.com/Integrated_RHG.html. Accessed: 12 August 2014. 202

[13] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 37–52, New York, NY, USA, 1993. ACM. 5

[14] Maribel Fernández. Type assignment and termination of interaction nets. *Mathematical Structures in Computer Science*, 8(6):593–636, 1998. 7

[15] Maribel Fernández and Ian Mackie. From term rewriting to generalised interaction nets. In H. Kuchen and S. D. Swierstra, editors, *Proceedings of the 8th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, number 1140 in Lecture Notes in Computer Science, pages 319–333. Springer-Verlag, September 1996. 137

[16] Maribel Fernández and Ian Mackie. Interaction nets and term rewriting systems. *Theoretical Computer Science*, 190(1):3–39, January 1998. 5

[17] Maribel Fernández and Ian Mackie. A calculus for interaction nets. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, number 1702 in LNCS, pages 170–187. Springer-Verlag, September 1999. 6, 7, 11, 13, 18, 29, 30, 31, 40

[18] Maribel Fernández and Ian Mackie. Operational equivalence for interaction nets. *Theoretical Computer Science*, 297(1–3):157–181, February 2003. 7

[19] Maribel Fernández, Ian Mackie, and Jorge Sousa Pinto. Combining interaction nets with externally defined programs,. In *Electronic proceedings of the APPIA-GULP-PRODE Joint Conference on Declarative Programming*, 2001. http://hdl.handle.net/1822/776. 149

[20] S. J. Gay. Interaction nets. Diploma in computer science, University of Cambridge Computer Laboratory, 1991. 7, 30

[21] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987. 1

[22] Jean-Yves Girard. Linear logic : its syntax and semantics. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, number 222 in London Mathematical Society Lecture Note Series, pages 1–42. Cambridge University Press, 1995. 2

[23] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989. 22

[24] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, January 1992. 5

[25] David Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, Department of Computer Science. University of Arizona, 1993. 171

[26] Abubakar Hassan. *Interaction Nets: Language Design and Implementation*. PhD thesis, University of Sussex, 2009. 7

[27] Abubakar Hassan, Eugen Jiresch, and Shinya Sato. An implementation of nested pattern matching in interaction nets. In Ian Mackie and Anamaria Martins Moreira, editors, *RULE*, volume 21 of *EPTCS*, pages 13–25, 2009. 10

[28] Abubakar Hassan, Ian Mackie, and Shinya Sato. Interaction nets: programming language design and implementation. *ECEASST*, 10, 2008. 9, 30, 31, 32, 199

[29] Abubakar Hassan, Ian Mackie, and Shinya Sato. Compilation of interaction nets. *Electr. Notes Theor. Comput. Sci.*, 253(4):73–90, 2009. 7, 9, 30, 31, 32, 199

[30] Abubakar Hassan, Ian Mackie, and Shinya Sato. A lightweight abstract machine for interaction nets. *ECEASST*, 29, 2010. 9, 30, 31, 40, 50, 62, 188

[31] Abubakar Hassan, Ian Mackie, and Shinya Sato. An implementation model for interaction nets. In Aart Middeldorp and Femke van Raamsdonk, editors, *Proceedings*

*8th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2015, Vienna, Austria, July 13, 2014.*, volume 183 of *EPTCS*, pages 66–80, 2015. 10

[32] Abubakar Hassan and Shinya Sato. Interaction nets with nested pattern matching. *Electr. Notes Theor. Comput. Sci.*, 203(1):79–92, 2008. 10

[33] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, July 2005. http://www.jucs.org/jucs_11_7/the_implementation_of_lua. 130

[34] Eugen Jiresch. Towards a gpu-based implementation of interaction nets. In Benedikt Löwe and Glynn Winskel, editors, *DCM*, volume 143 of *EPTCS*, pages 41–53, 2014. 7, 30, 31, 40

[35] S.L. Peyton Jones. *The Implementatiion of Functional Programming Languages*. Prentice-Hall International, 1987. 179

[36] Yves Lafont. Interaction nets. In *Seventeenth Annual Symposium on Principles of Programming Languages*, pages 95–108, San Francisco, California, 1990. ACM Press. 3, 5, 11, 12, 13, 16, 29, 30, 31

[37] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, January 1964. 51, 76

[38] Sylvain Lippi. $\text{in}^2$ : A graphical interpreter for interaction nets. In Sophie Tison, editor, *RTA*, volume 2378 of *Lecture Notes in Computer Science*, pages 380–386. Springer, 2002. 6, 30, 31, 32

[39] Sylvain Lippi. *Théorie et pratique des réseaux d'interaction*. PhD thesis, Université de la méditerranée, 2002. 5, 7

[40] Sylvain Lippi. The graphical krivine machine. *Higher-Order and Symbolic Computation*, 20(3):295–318, 2007. 5, 6

[41] Ian Mackie. *The Geometry of Implementation*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1994. 5

[42] Ian Mackie. Linear logic *with* boxes. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS'98)*, pages 309–320. IEEE Computer Society Press, June 1998. 4

[43] Ian Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998. 5, 19, 64

[44] Ian Mackie. Interaction nets for linear logic. *Theoretical Computer Science*, 247(1):83–140, September 2000. 4

[45] Ian Mackie. Towards a programming language for interaction nets. *Electronic Journal in Theoretical Computer Science*, 127(5):133–151, May 2005. 4, 6, 7

[46] Ian Mackie. Encoding strategies in the lambda calculus with interaction nets. In Andrew Butterfield, editor, *Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages (IFL'05)*, volume 4015 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006. 4, 5

[47] Ian Mackie. A rewriting paradigm for program and algorithm animation. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009, Corvallis, OR, USA, 20-24 September 2009, Proceedings*, pages 170–173. IEEE Computer Society, 2009. 6

[48] Ian Mackie. A visual model of computation. In J. Kratochvil et al., editor, *Theory and Applications of Models of Computation, 7th Annual Conference, TAMC 2010, Prague, Czech Republic*, volume 6108 of *Lecture Notes in Computer Science*, pages 350–360. Springer-Verlag, June 2010. 6

[49] Ian Mackie and Shinya Sato. A calculus for interaction nets based on the linear chemical abstract machine. *Electr. Notes Theor. Comput. Sci.*, 192(3):59–70, 2008. 14, 69

[50] Ian Mackie and Shinya Sato. An interaction net encoding of Gödel's System T. In *Pre-Proceedings of the Fifth International Workshop on Graph Computation Models*, to appear. 9

[51] Ian Mackie and Shinya Sato. Some observations for the parallel implementation of interaction nets. In *Pre-Proceedings of the 10th International Workshop on Developments in Computational Models*, to appear. 10

[52] Damiano Mazza. *Interaction Nets: Semantics and Concurrent Extensions*. Ph.D. thesis, Université de la Méditerranée/Università degli Studi Roma Tre, 2006. 7

[53] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM. 186

[54] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. 7, 187

[55] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003. 7

[56] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report 476, Indiana, 1997. 7

[57] Jorge Sousa Pinto. Sequential and Concurrent Abstract Machines for Interaction Nets. In Jerzy Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, number 1784 in Lecture Notes in Computer Science, pages 267–282. Springer-Verlag, 2000. 7, 30, 40, 49, 50, 72

[58] Jorge Sousa Pinto. *Parallel Implementation with Linear Logic*. PhD thesis, École Polytechnique, February 2001. 4, 6, 7

[59] 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. 7, 148

[60] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 3rd edition, 2009. 170

[61] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011. 187

# Appendix A

# Programs in related works

## A.1 amineLight: runtime functions

```
void eval() {
  Agent *a1, *a2;
  while (popActive(&a1, &a2)) {
    if (a2->id != ID_NAME) {
      if (a1->id != ID_NAME) {
        R[a1][a2](a1, a2);
      } else {
        /* operations for x=Alpha(x1,...,xn) */
        if (a1->port[0] == NULL) {
          /* II.0 */
          a1->port[0] = a2;
        } else if ((a1->port[0])->id != ID_NAME) {
          /* II.e */
          Agent *a1p0 = a1->port[0];
          freeAgent(a1);
          a1=a1p0;
          pushActive(a1,a2);
        } else {
          /* II.c */
          (a1->port[0])->port[0] = a2;
          freeAgent(a1);
        }
```

```
  }
} else {
  /* operations for Alpha(x1,...,xn)=y and x=y */
  if (a1->id != ID_NAME) {
    /* II.- */
    if (a2->port[0] == NULL) {
      a2->port[0] = a1;
    } else if ((a2->port[0])->id != ID_NAME) {
      Agent *a2p0 = a2->port[0];
      freeAgent(a2);
      a2=a2p0;
      pushActive(a1,a2);
    } else {
      (a2->port[0])->port[0] = a1;
      freeAgent(a2);
    }
  } else {
    if (a1->port[0] == NULL) {
      if (a2->port[0] == NULL) {
        /* III.0_0 */
        a1->port[0] = a2;
        a2->port[0] = a1;
      } else if ((a2->port[0])->id != ID_NAME) {
        /* III.0_e */
        a1->port[0] = a2->port[0];
        freeAgent(a2);
      } else {
        /* III.0_c */
        a1->port[0] = a2->port[0];
        (a2->port[0])->port[0] = a1;
        freeAgent(a2);
      }
    } else if ((a1->port[0])->id != ID_NAME) {
      if (a2->port[0] == NULL) {
```

```
    /* III.e_0 */
    a2->port[0] = a1->port[0];
    freeAgent(a1);
  } else if ((a2->port[0])->id != ID_NAME) {
    /* III.e_e */
    Agent *a1p0=a1->port[0];
    freeAgent(a1);
    Agent *a2p0=a2->port[0];
    freeAgent(a2);
    a1=a1p0;
    a2=a2p0;
    pushActive(a1,a2);
  } else {
    /* III.e_c */
    (a2->port[0])->port[0] = a1->port[0];
    freeAgent(a1);
    freeAgent(a2);
  }
} else {
  if (a2->port[0] == NULL) {
    /* III.c_0 */
    (a1->port[0])->port[0] = a2;
    a2->port[0] = a1->port[0];
    freeAgent(a1);
  } else if ((a2->port[0])->id != ID_NAME) {
    /* III.c_e */
    (a1->port[0])->port[0] = a2->port[0];
    freeAgent(a1);
    freeAgent(a2);
  } else {
    /* III.c_c */
    (a2->port[0])->port[0] = a1->port[0];
    (a1->port[0])->port[0] = a2->port[0];
    freeAgent(a1);
```

```
            freeAgent(a2);
          }
        }
      }
    }
  }
}
```

# Appendix B

# Benchmark programs

In this chapter, we show the souce files of benchmark programs on interger numbers and lists in SML, Python and Inpla.

## B.1 Ackermann function

**SML**

```
fun ack 0 n = n+1
  | ack m 0 = ack (m-1) 1
  | ack m n = ack (m-1) (ack m (n-1));

ack 3 9;
```

**Python**

```
def ack(m,n):
    if m==0:
        return n+1
    elif n==0:
        return ack(m-1, 1)
    else:
        return ack(m-1, ack(m, n-1))

print ack(3,9)
```

**Inpla**

```
A(n,r) >< (int m)
  | m==0 => Addn(1,r)~n
  | _ => A2(m, r)~n;

A2(int m,r) >< (int n)
  | n==0 => A((1),r)~(m1) where m1=m-1
  | _ => A(w,r)~(m1), A((n1),w)~(m) where n1=n-1 m1=m-1;

Addn(int n, r) >< (int m)=> r~(i) where i=n+m;

A((9),r)~(3);
r;
```

## B.2 Fibonacci number

**SML**

```
fun fib 0 = 1
  | fib 1 = 1
  | fib n = (fib (n-1)) + (fib (n-2));

fib 39;
```

**Python**

```
def fib(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

print fib(39)
```

**Inpla**

```
Fib(r) >< (int a)
  | a == 0 => r~(1)
  | a == 1 => r~(1)
  | _ => Fib(n1)~(b),Fib(n2)~(c), Add(n2,r)~n1
    where b=a-1 c=a-2;

Add(n2,r) >< (int i)
=> Add2(i, r) ~ n2;

Add2(int i, r) >< (int j)
=> r~(a) where a=i+j;

Fib(r)~(39);
r;
```

## B.3 Bubble sort

### SML

http://rosettacode.org/wiki/Sorting_algorithms/Bubble_sort#OCaml

```
local
    fun bsortsub (x::x2::xs) =
          if x > x2 then x2::(bsortsub (x::xs))
          else x::(bsortsub(x2::xs))
      | bsortsub x = x;
in
    fun bsort x =
    let
      val s = bsortsub x;
    in
      if x=s then x else bsort s
    end
end;

(* mkRandList *)
local
```

```
    val nextInt = Random.randRange(1,10000);
    val r = Random.rand(1,1);
in
  fun mkRandList 0 = []
    | mkRandList n = (nextInt r)::(mkRandList (n-1))
end;

bsort (mkRandList 20000);
```

### Python

http://www.geekviewpoint.com/python/sorting/bubblesort

```
import random

def mkRandList ( n ):
    a=[]
    for i in range(1,n+1):
        a.insert(0, random.randint(0,10000))
    return a

def bubblesort( A ):
    for i in range( len( A ) ):
        for k in range( len( A ) - 1, i, -1 ):
            if ( A[k] < A[k - 1] ):
                tmp = A[k]
                A[k] = A[k-1]
                A[k-1] = tmp

a = mkRandList(20000)
bubblesort(a)
```

### Inpla

```
BS(r) >< [] => r~[];
BS(r) >< [x | xs] => B(x, BS(r))~xs;
BS(r) >< M(w) => r~w;
```

```
B(int x,r) >< [] => r~M([x]);
B(int x,r) >< M(w) => r~M([x | w]);
B(int x,r) >< [int y | ys]
  | x<y => r~[x|w], B(y,w)~ys
  | _   => r~[y|w], B(x,w)~ys;

MkRandList(r) >< (int n)
  | n>0 => r~[rd|r1], MkRandList(r1)~(n1)
    where n1=n-1 rd=rand(10000)
  | _ => r~[];

MkRandList(r)~(20000), BS(r1)~r;
r1;
```

## B.4 Quicksort

### SML

http://www.webber-labs.com/mpl/source%20code/Chapter%20Twelve/quicksort.sml.txt

```
fun quicksort nil = nil
  | quicksort (pivot :: rest) =
    let
        fun split(nil) = (nil,nil)
          | split(x :: xs) =
          let
            val (below, above) = split(xs)
          in
            if x < pivot then (x :: below, above)
            else (below, x :: above)
          end;
          val (below, above) = split(rest)
    in
        quicksort below @ [pivot] @ quicksort above
    end;
```

```
(* mkRandList *)
local
    val nextInt = Random.randRange(1,10000);
    val r = Random.rand(1,1);
in
  fun mkRandList 0 = []
    | mkRandList n = (nextInt r)::(mkRandList (n-1))
end;

quicksort (mkRandList 500000);
```

### Python

http://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Python

```
import random

def quickSort(arr):
    less = []
    pivotList = []
    more = []
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        for i in arr:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)
                less = quickSort(less)
                more = quickSort(more)
                return less + pivotList + more
```

```
def mkRandList ( n ):
    a=[]
    for i in range(1,n+1):
        a.insert(0, random.randint(0,10000))
    return a

a = mkRandList(500000)
quickSort(a)
```

**Inpla**

```
QS(r) >< [] => r~[];
QS(r) >< [int x|xs] => Part(x, QS(w), QS(App([x|w], r)))~xs;

App(a,b) >< [] => a~b;
App(a,b) >< [x|xs] => b~[x|w], xs~App(a,w);

Part(int x, a, b) >< [] => a~[], b~[];
Part(int x, a,b) >< [int y|ys]
  | y<x => ys~Part(x, a, w), b~[y|w]
  | _   => ys~Part(x, w, b), a~[y|w];

MkRandList(r) >< (int n)
  | n>0 => r~[rd|r1], MkRandList(r1)~(n1)
    where n1=n-1 rd=rand(10000)
  | _ => r~[];

MkRandList(r)~(500000), QS(r1)~r;
r1;
```