Using Logic for Concurrency: A Critical Study

Adolfo Gustavo Serra Sêca Neto

Orientador: Ruy José Guerra Barretto de Queiroz

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE CIÊNCIAS EXATAS E DA NATUREZA

DEPARTAMENTO DE INFORMÁTICA

Using Logic for Concurrency: A Critical Study

Adolfo Gustavo Serra Sêca Neto

Monografia apresentada como parte dos requisitos
para obtenção do título de Mestre em Ciência da
Computação.

Orientador: Ruy José Guerra Barretto de Queiroz

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE CIÊNCIAS EXATAS E DA NATUREZA
DEPARTAMENTO DE INFORMÁTICA

Recife, 13 de dezembro de 1996

FICHA CATALOGRÁFICA

Using Logic for Concurrency: A Critical Study

Adolfo Gustavo Serra Sêca Neto

Orientador: Ruy José Guerra Barretto de Queiroz

Dissertação de Mestrado

Mestrado em Ciência da Computação,

Departamento de Informática,

Universidade Federal de Pernambuco

Apresentada em 13 de dezembro de 1996

Agradecimentos

Ao professor Ruy de Queiroz.

Aos professores Benedito Acioly e Edward Hermann Hauesler, pelas críticas e sugestões feitas ao trabalho.

Aos todos os professores da pós-graduação de Departamento de Informática da UFPE.

Aos colegas professores da UFAL, em especial a Evandro, Fábio Paraguaçú e Cid Albuquerque.

Aos colegas da pós-graduação, em especial a Adriano Augusto, Ticiano, Luis Sérgio, Thalles, Babacar, Fred, Daniela, Leonardo, Márcia, Nélson, Alcione, Jeanne, Liliane, Adriano Lorena, Ricardo Massa, Ricardo Salgueiro, Edylaine, Genésio.

Aos meus pais, Albino e Lídice e meus irmãos Albino Jr. e Antônio Augusto pelo apoio e encorajamento.

Resumo

Concorrência—o estudo da teoria e da prática de sistemas concorrentes—é um assunto muito importante em computação atualmente. Sistemas concorrentes são utilizados em diversas aplicações, por exemplo, no projeto de protocolos de redes de computadores e na modelagem de transações em bancos de dados. Apesar dos avanços ocorridos nesta área nos últimos anos, com o desenvolvimento de vários novos modelos de concorrência (ex. CCS, CSP, π-cálculo, μ-cálculo), pode-se dizer que ainda existem alguns problemas em aberto nesta área. Por exemplo, a inexistência de bons cálculos básicos tipados para formalização e raciocínio sobre propriedades de processos concorrentes. Alguns trabalhos recentes tentaram relacionar sistemas lógicos a modelos matemáticos de concorrência com o intuito de fornecer uma base lógica para a teoria de concorrência e prover meios para a resolução de problemas através de métodos formais. Os trabalhos neste sentido que apresentaram melhores resultados foram aqueles que relacionaram a lógica linear (desenvolvida por Jean-Yves Girard em 1987) ao π-cálculo, um modelo algébrico de concorrência desenvolvido por Robin Milner.

Nesta dissertação analisamos as principais abordagens para a obtenção de uma base lógica para concorrência. Em primeiro lugar, discutimos em detalhes lógica linear a fim de entender os trabalhos que a utilizam para explicar concorrência. Em seguida, estudamos os modelos algébricos de processos concorrentes (com destaque para CCS e π-cálculo) e algumas de suas principais características. Vimos então como os trabalhos que utilizam lógica linear se enquadram no contexto daqueles que relacionam lógica e concorrência. Os trabalhos com lógica linear baseiam-

se no paradigma ‘provas como processos’ de Samson Abramsky, uma variação do paradigma já bastante estabelecido ‘proposições como tipos’ para o mundo da concorrência. O paradigma ‘proposições como tipos’, por sua vez, é a base da bem-sucedida Interpretação Funcional de Curry-Howard (que fornece uma base lógica e um sistema de tipos para programação funcional, relacionando-a à lógica intuicionista). Apesar de não ter sido completamente bem-sucedido com relação aos objetivos propostos por Abramsky, o trabalho de Gianluigi Bellin e Phil Scott sobre a correspondência entre a lógica linear e o π-cálculo obteve resultados significativos e serve como base para futuros trabalhos relacionando lógica e concorrência.

Por fim, discutimos sucintamente a possibilidade de utilizar os Sistemas Dedutivos Rotulados de Dov Gabbay com o objetivo semelhante de fornecer uma base lógica para concorrência baseada na Interpretação Funcional de Curry-Howard.

Palavras-chave: concorrência, lógica, lógica linear, CCS, π-cálculo, LDS. æ

Abstract

Concurrency—the study of the theory and practice of concurrent systems—is a very important subject in computer science nowadays. Concurrent systems are used in several applications such as, for example, the design of computer network protocols and the modelling of database transactions. Although there has been several advances in this area with the development of new models of concurrent behaviour (e.g. CCS, CSP, π-calculus and μ-calculus), one can say that some problems remain to be solved. For instance, there are not good basic typed calculi to formalize concurrency properties. Some recent works attempted to establish a correspondence between logical systems and mathematical models of concurrency; they aimed at providing a logical basis for concurrency theory and at solving problems by using formal methods. The better results obtained in this approach were those achieved by the works that relate linear logic (developed by Jean-Yves Girard in 1987) to the π-calculus, an algebraic model of concurrency designed by Robin Milner.

In this dissertation we have analysed the main approaches used to obtain a logical basis for concurrency. First, we have discussed in detail linear logic in order to understand the works that use it to explain some concurrency features. After that, we have studied algebraic models of concurrency (especially CCS and π-calculus) emphasizing some of their main features. Then we have seen how the works that use linear logic can be put in the framework of those works that relate logic and concurrency. Samson Abramsky's 'proofs as processes' paradigm is the foundation for the papers using linear logic. It is an adaptation of the well-established

'propositions as types' paradigm to the concurrency world. The propositions as types paradigm, by the way, is the foundation to the successful Curry-Howard functional interpretation, which provides a logical basis and a type system for functional programming by relating it to intuitionistic logic. Although it has not been completely successful in regard to the objectives proposed by Abramsky, Gianluigi Bellin and Phil Scott's work about the relationship between linear logic and π-calculus has achieved significant results and may inspire future works relating logic to concurrency.

We finish by discussing quite succintly the possibility of using Dov Gabbay's Labelled Deductive Systems (LDS) with the similar objective of providing a logical basis for concurrency based on Curry-Howard functional interpretation.

Keywords: concurrency, logic, linear logi, CCS, π-calculus, LDS. æ

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Concurrent systems are very important nowadays. When we use a teller machine, a cellular telephone or the Internet, for example, we are interacting with systems with a concurrent behaviour. We say that concurrent systems are those systems where one or more actions can be executed at the same time. Besides that, the processing of some concurrent systems may be *spatially distributed,* that is, different processing units in distinct places are in charge of parts of the system's processing. Also, many of the concurrent systems in practice are also *reactive systems,* i.e. they do not perform a specific set of actions in a given order but rather react to stimuli presented by the users and the working environment. Concurrency is the area that studies the theory and the practice of concurrent systems. For the reasons stated above, one can see that this is a very important subject in computer science.

There has been several advances in the two last decades regarding concurrency. For example, in 1980 Robin Milner presented his Calculus of Communicating Sys-

tems (CCS), a calculus where concurrent processes can be described as terms of an algebra (e.g. $P \parallel Q$ represents the parallel composition of two processes [Mil80]). In 1992, Milner, Parrow and Walker refined CCS and obtained a calculus of mobile processes, the π-calculus. Notwithstanding these and many other advances towards the development of calculi for the representation of concurrency, several models of concurrent behaviour still present some problems. These problems seem serious when these models are compared to sequential and functional models of computation. For example, there is no adequate typing discipline for concurrent processes and no normal form theorems for descriptions of concurrent processes in most calculi. A solution to these problems would make models of concurrency more useful as formal methods for specifying concurrent systems and concurrent systems specification languages.

In order to try to solve these problems, a possibility is to formulate *logical systems* to reason about concurrent processes' descriptions. The relationship between formulas and processes presented by these systems will hopefully provide insights that might help to discover, among other things, what is the most adequate equivalence relation or the structure of normal processes (processes written in a special, somehow simplified form). In [Mil93], for example, Milner defines a modal logic "to give an alternative characterization of the bisimilarity relations in π-calculus". In [AD96], the idea was to obtain a system that could automatically prove process properties. Therefore, there are several works in the literature relating logic and mathematical models of concurrency in order to solve concurrency problems.

Here, we will discuss some of the approaches used to perform such relationship. The most important of the approaches presented here is, in our opinion, Abramsky's

'proofs as processes' paradigm. The intention in this paradigm is to adapt the well-known 'propositions as types' paradigm to the concurrency world. In the works that follow this approach, process descriptions are associated to formulas or proofs in linear logic. In this way, the formulas or proofs serve as processes' types and some of the processes' properties may be logically represented. One of the applications of these types would be to restrict the composition of processes. For example, if the types of two processes indicate that their composition may enter in a deadlock state, then the type system must prevent this composition because of the processes' types.

The works [Abr93, BS94] use this paradigm to perform the interaction between *linear logic* and a model of concurrency. One of linear logic main features is its resource consciousness. That is, the use of each formula alongside proofs is accounted; the system is not indiferent to the number of times each formula is used, as it happens in classical and intuitionistic logic, for example. This resource accounting may be important to represent computer science applications since in this way the use of computational resources (e.g. memory, disks, processors) can be represented. We shall see that other features also suggest that linear logic can be used to represent computer science applications; this has been demonstrated by several works in the literature.

Besides that, we will present some features of mathematical models of concurrency that must be taken into account to understand the logical interpretation of process descriptions. For instance, one of these features is mobility, which is related to the possibility of a process changing its environment. This and other features will be discussed and we will also present two important models of concurrency in the

algebraic approach: Robin Milner's Calculus of Communicating Systems (CCS) and π-calculus. Both calculi were used in works relating logic and concurrency.

Later we will discuss the possibility of using Dov Gabbay's Labelled Deductive Systems (LDS) aiming at results similar to those seeked by the works analysed here, i.e. to provide a logical foundation to concurrency. We believe that LDS is a good tool for this kind of work for two main reasons. First, LDS is a general framework for the definition of logical systems—therefore, once the logical system is not committed to any particular logic, one is free to represent several features of concurrency. Second, LDS's declarative unit is the labelled formula $(t : \mathbf{A})$, where $\mathbf{A}$ is a formula and t is a label. In the label one can record several considerations performed at the meta-level in other logical systems—thus, we think that we can use this feature to define a logical system for concurrency where the formulas remain as close as possible to classical logic formulas, and the complexity is handled in the labels. Some observations towards the definition of such a system were presented here.

1.1 Outline

First, in Chapter 2 we will present Girard's linear logic, a new logical system developed in 1987 that has been widely accepted in the computer science theory research community. Next, in Chapter 3 we will study some features of models of concurrent behaviour and discuss in some detail two algebraic models of concurrency: CCS and π-calculus. Having studied linear logic and algebraic models of concurrency we will be able to discuss the works that try to establish a corre-

spondence between logic and concurrency in Chapter 4. The main discussion is on the relationship between linear logic and the π-calculus; these works follow the 'proofs as processes' paradigm. We will show that Bellin and Scott's paper on the relationship between the π-calculus and linear logic contains the most important results in this approach. In Chapter 5 we will briefly discuss how we could use Gabbay's Labelled Deductive Systems in order to develop a logical system to reason about concurrent processes. Finally, in Chapter 6 we will discuss the results of our analysis and present some possibilities for future works.

æ

Chapter 2

Linear Logic

2.1 Introduction

Linear logic is a *new* logical system, with new logical connectives and a proof theory very different from classical logic proof theory. Its acceptance by computer science theory research community has been so impressive that several works have appeared either studying linear logic by itself or relating it to several other subjects in computation, such as concurrency and programming languages. Linear logic was first presented in 1987 and since then it has been much discussed and also extended in many forms. However, because of the factors discusssed in Section 2.6, linear logic does not seem to be the most adequate logical tool for the representation of computer science applications. Here we are going to present the *main* features of linear logic in sufficient detail to understand the works relating it to concurrency, which are the subject of Chapter 4.

2.1.1 Motivation

Linear logic has appeared in 1987 and by this time several other studies on the creation of logical systems whose main objective was *also* to give meaning to computation aspects had already appeared (see [Gab90] about the area called 'Logic and Computation'). Some of these studies resulted in very interesting systems (e.g. Martin-Loef's Intuitionistic Type Theory [Mar84]) which contributed to a better understanding of the foundations of computer science. Since then many other logical formalisms for computation have been presented (such as Gabbay's LDS, which we will discuss in Chapter 5) and linear logic has been established as one of the most used formalisms for the study of the interface between logic and computation.

2.2 Features of the System

The proof-theoretically driven development of linear logic led Girard to incorporate into linear logic some features that where also important for a logic that deals with aspects of computation, such as:

(i) the possibility of interpreting a sequent as the state of a system;

(ii) the treatment of a formula as a resource and;

(iii) the acceptance of 'double negation' without losing constructivity.

In the following we analyse these features and in Figure 2.1 we show a table summarizing this analysis.

Features	**Importance**	**Consequences**
States and transitions	Classical logic does not represent these notions directly	Proofs as actions
	Computer science applications demand these notions	theory = linear logic + axioms + current state
Resource awareness	Representation of the use of computational or other kind of resources	Each formula must be used exactly once in a derivation
Double (linear) negation	Constructiveness of the system	A proof of $\exists \mathbf{x}\mathbf{A}[\mathbf{x}]$ contains a proof $\mathbf{A}[t]$ for some t. A proof of $\mathbf{A} \oplus \mathbf{B}$ contains a proof of $\mathbf{A}$ or a proof of $\mathbf{B}$.

Table 2.1: Summary of linear logic features

2.2.1 States and Transitions

The first interesting feature of linear logic that we are going to describe is the representation of states and transitions. Linear logic allows a representation of the notion of *state* of a system in a very satisfactory way. That is, it is possible to describe states of systems, as well as *transitions* between these states, by using linear logic to formalize these systems. Linear logic achieves this mainly because of the resource awareness of its connectives and of its proof theory (see more in subsection 2.2.2); it does not have an extra apparatus in order to represent these notions. Let us see an example. The well-known chemical formula describing the formation of molecules of water:

$$2H_2 + O_2 \rightarrow 2H_2O$$

denotes that there is an *action* that *consumes* two molecules of hydrogen and one of oxygen in order to *produce* two molecules of water. The *reaction* to this action, which is implicit in the equation, is that the molecules of hydrogen and oxygen can no longer be used to form other molecules. In this way, the state of the system in the lefthand side of the equation ($2H_2 + O_2$) is changed to the state in the righthand side ($2H_2O$) by the 'transition' ($\rightarrow$) that produces molecules of water.

Using linear implication ($\multimap$) and linear multiplicative conjunction ($\otimes$) it is possible to represent the above equation by the following formula [Gir95b]:

$$\mathbf{H_2} \otimes \mathbf{H_2} \otimes \mathbf{O_2} \multimap \mathbf{H_2O} \otimes \mathbf{H_2O}$$

where multiplicative conjunction is used in order to join the molecules that take part in a state, whilst implication represents the transition that takes the system

from one state to another. Since linear implication is the meaning of the derivability sign ($\vdash$) in the linear sequent calculus, this feature holds for any linear logic proof system.

Importance

This feature of linear logic is important for two reasons. The first reason is that in classical logic it is *not* possible to give an adequate representation of states and transitions. According to Girard [Gir95b], this has occurred because of the 'excessive focusing of logicians in mathematics'; in classical logic, it is only possible to represent states and transitions by introducing an extraneous temporal parameter. For example, the transition $S \rightarrow S'$ can be represented in classical logic by two propositions, (S, t) and $(S, t+1)$, where (S, t) means that S holds at instant t. This happens because in classical logic the propositions represent *stable truths:* a formula once true remains true forever, or, in Artificial Intelligence terms, one cannot 'remove a fact from the knowledge basis'. Non-monotonic logics tried to solve this problem but the solutions found were considered inadequate by Girard. Therefore, linear logic is a new approach towards solving the problem of describing states and transitions in a logical system.

The second reason for the importance of this representation is that these two concepts are very important for computer science. Many computational systems can be represented by using these notions and the fact that linear logic supports them is truly an advantage for its application in computer science. As an example of this importance, consider the representation of a database querying system. Suppose that this system is asked to perform a certain query and, after the query

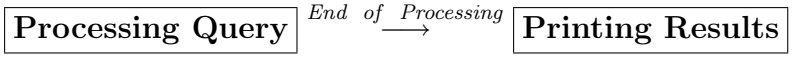

Figure 2.1: Transition in a database querying system

is processed, it prints the results found. Therefore, while the system is processing the query it remains in a given state. After the processing ends, the system will begin to print the results, entering in another state. The transition that takes the system from one state to another is the end of the processing of the query. This is pictured in Figure 2.1, where the states are named **Processing Query** and **Printing Results**, whilst the transition is called *End of Processing*. To be able to represent these notions is very important for any attempt to formalize several kinds of computational systems. For concurrent systems in particular, which are going to be discussed in Chapter 3, the notions of states and transitions play an even more important role than in sequential systems (see [Gup94, WN95a]).

Consequences

There are two major consequences of this feature to linear logic proof theory. First, the proofs can be seen as actions (i.e. transitions) which modify the states of systems. If the states are represented by linear logic sequents, the following transitions, for example,

$$E_1 \xrightarrow{t_1} E_2 \xrightarrow{t_2} E_3$$

would be represented in a proof as

$$\frac{\delta(t_1) \quad \dfrac{\delta(t_2) \quad \delta(E_3)}{\delta(E_2)}}{\delta(E_1)}$$

where $\delta(E_i)$ and $\delta(t_j)$ are linear logic sequents, according to a translation $\delta()$ defined for the states E_i and transitions t_j. This kind of translation can be seen, for example, in the works studying the computational complexity of linear logic fragments, such as [Lin92a].

The second consequence for proof theory is that a new definition of the notion of *theory* is needed. In classical logic, a theory is *classical logic + axioms*. In linear logic, due to the representation of states, a theory must be redefined as *linear logic + axioms + current state*. That is, in linear logic the current state of the logical system, which is a multiset of formulas (see subsection 2.3.1), affects the set of formulas that can be proved from a given theory. For example, for a theory composed of a single axiom, $\mathbf{A} \multimap \mathbf{B}$, the two following *different* theories (T_1 and T_2) can be defined:

$$T_1 = (\text{axioms} = \frac{}{\mathbf{A} \vdash \mathbf{B}} T_1, \text{current state} = \mathbf{A})$$

$$T_2 = (\text{axioms} = \frac{}{\mathbf{A} \vdash \mathbf{B}} T_2, \text{current state} = \mathbf{A} \otimes \mathbf{A})$$

And T_1 can prove $\mathbf{B}$, but not $\mathbf{B} \otimes \mathbf{B}$ (because the formula $\mathbf{A}$ in T_1's current state can be used only once), whilst T_2 can prove $\mathbf{B} \otimes \mathbf{B}$.

Another important feature of linear logic theories is that the axioms can be replaced by exponentiated formulas such as $!\mathbf{A}$ [Lin92a]. That is, the axioms in a theory can be represented as a set of exponentiated formulas instead of a set of axiom rules.

These exponentiated formulas will appear in the lefthand side of the conclusion sequent in any proof in that theory. In this way, the following proof in theory T_2

$$\dfrac{\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{B}}T_2 \quad \dfrac{}{\mathbf{A} \vdash \mathbf{B}}T_2}{\mathbf{A}, \mathbf{A} \vdash \mathbf{B} \otimes \mathbf{B}} \otimes R}{\mathbf{A} \otimes \mathbf{A} \vdash \mathbf{B} \otimes \mathbf{B}} \otimes L$$

would be transformed into

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I \quad \dfrac{}{\mathbf{B} \vdash \mathbf{B}}I}{\mathbf{A} \multimap \mathbf{B}, \mathbf{A} \vdash \mathbf{B}} \multimap L}{!(\mathbf{A} \multimap \mathbf{B}), \mathbf{A} \vdash \mathbf{B}}!D \quad \dfrac{\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I \quad \dfrac{}{\mathbf{B} \vdash \mathbf{B}}I}{\mathbf{A} \multimap \mathbf{B}, \mathbf{A} \vdash \mathbf{B}} \multimap L}{!(\mathbf{A} \multimap \mathbf{B}), \mathbf{A} \vdash \mathbf{B}}!D}{!(\mathbf{A} \multimap \mathbf{B}), !(\mathbf{A} \multimap \mathbf{B}), \mathbf{A}, \mathbf{A} \vdash \mathbf{B} \otimes \mathbf{B}} \otimes R}{!(\mathbf{A} \multimap \mathbf{B}), !(\mathbf{A} \multimap \mathbf{B}), \mathbf{A} \otimes \mathbf{A} \vdash \mathbf{B} \otimes \mathbf{B}} \otimes L}{!(\mathbf{A} \multimap \mathbf{B}), \mathbf{A} \otimes \mathbf{A} \vdash \mathbf{B} \otimes \mathbf{B}}!C$$

In this latter proof one can see that the axiom $\dfrac{}{\mathbf{A} \vdash \mathbf{B}}T_2$ is replaced by the formula $!(\mathbf{A} \multimap \mathbf{B})$ in the last sequent. That is, axioms such as $\mathbf{\Gamma} \vdash \mathbf{\Delta}$ are transformed into exponentiated formulas such as $!((\otimes\mathbf{\Gamma}) \multimap (\wp\mathbf{\Delta}))$[1] and put in the last sequent. Then, one tries to prove conclusions (the formulas in the righthand side of the sequent) from the current state of the theory. For example, the axiom $\dfrac{}{\mathbf{A}, \mathbf{B} \vdash \mathbf{C}, \mathbf{D}}$ would be tranlated into $!(\mathbf{A} \otimes \mathbf{B}) \multimap (\mathbf{C} \wp \mathbf{D})$. This idea works because exponentiated formulas can be used any number of times, just like axioms in theories.

2.2.2 Resource Awareness

According to Girad, in classical as well as in intuitionistic logic, propositions are considered *stable truths.* Therefore, one cannot represent (at least directly) the

[1] $\wp$ is linear multiplicative disjunction.

$$\dfrac{\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}} I \quad \dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}} I \quad \dfrac{}{\mathbf{B} \vdash \mathbf{B}} I}{\mathbf{A}, \mathbf{A} \Rightarrow \mathbf{B} \vdash \mathbf{B}} \Rightarrow L}{\mathbf{A}, \mathbf{A}, \mathbf{A} \Rightarrow \mathbf{B} \vdash \mathbf{A} \wedge \mathbf{B}} \wedge R}{\boxed{\mathbf{A}}, \mathbf{A} \Rightarrow \mathbf{B} \vdash \mathbf{A} \wedge \mathbf{B}} \text{Contraction L}$$

Figure 2.2: Classical logic inference

$$\dfrac{\dfrac{\boxed{a : \mathbf{A}} \quad x : \mathbf{A} \Rightarrow \mathbf{B}}{ax : \mathbf{B}} \Rightarrow E \qquad \dfrac{\boxed{a : \mathbf{A}} \quad y : \mathbf{A} \Rightarrow \mathbf{C}}{ay : \mathbf{C}} \Rightarrow E}{< ax, ay >: \mathbf{B} \wedge \mathbf{C}} \wedge I$$

Figure 2.3: Intuitionistic natural deduction inference

use of a formula treated as a resource. In these logics, he notices, any formula can be used (i.e. can take part in an application of an inference rule) any number of times. This is illustrated in Figure 2.2, where the formula **A** is duplicated in the application of the Contraction rule and used twice: first to 'produce' **B** and then to 'produce' $\mathbf{A} \wedge \mathbf{B}$. In Figure 2.3, we use labels to mark instances of a formula. Different formulas and different instances of the same formula receive different labels. In the figure, the same formula ($a : \mathbf{A}$) is used twice: first to produce **B** and then to produce **C**. This is considered a valid inference in Intuitionistic Natural Deduction and all used formulas remain valid (i.e. can be further used in the deduction).

But linear logic is a *resource-aware* logic. This means that the validity of a deriva-

$$\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I \qquad \dfrac{}{\mathbf{B} \vdash \mathbf{B}}I}{\mathbf{A}, \mathbf{A} \multimap \mathbf{B} \vdash \mathbf{B}} \multimap L$$

Figure 2.4: Use of all resources

tion depends on how the formulas, which are the *resources* in a proof, are used along the proof. A linear logic formula, once used, is not available anymore for further use (see Figure 2.4, where $\mathbf{A}$ and $\mathbf{A} \multimap \mathbf{B}$[2] are both used in order to produce $\mathbf{B}$). Thus, if one wants to use a formula more than once one has to have other copies of it, where these other copies are treated as different *instances* of the same formula. This can be seen in Figures 2.5 and 2.6, where two different instances of the same formula ($\mathbf{A}$) are needed in order to produce the conclusion.

Actually, linear logic not only requires that a *used* formula is not available for further use but also that each formula be used *exactly once*. This means that *all* available resources *must* be used in a derivation. Therefore, the deduction in Figure 2.7 is not valid because only one of the $\mathbf{A}$'s ($a : \mathbf{A}$) was used in order to produce $\mathbf{B}$. However, the deduction in Figure 2.6 is valid because all instances of formulas were used in some inference in order to obtain $\mathbf{B} \otimes \mathbf{C}$.

This resource awareness of linear logic is interesting because in this way one can represent the use of computational resources (such as memory, disks, processors) in a computation. For example, one can count the number of accesses to memory in a

[2] $\mathbf{A} \multimap \mathbf{B}$ ($\mathbf{A}$ *linearly implies* $\mathbf{B}$) means that $\mathbf{A}$ *has* to be used *exactly once* in order to obtain $\mathbf{B}$. See Section 2.5 for further details.

$$\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I \qquad \dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I \qquad \dfrac{}{\mathbf{B} \vdash \mathbf{B}}I}{\mathbf{A}, \mathbf{A} \multimap \mathbf{B} \vdash \mathbf{B}} \multimap L}{\mathbf{A}, \mathbf{A}, \mathbf{A} \multimap \mathbf{B} \vdash \mathbf{A} \otimes \mathbf{B}} \multimap L$$

Figure 2.5: Two instances of the same formula

$$\dfrac{\dfrac{a : \mathbf{A} \quad x : \mathbf{A} \multimap \mathbf{B}}{ax : \mathbf{B}} \multimap E \qquad \dfrac{b : \mathbf{A} \quad y : \mathbf{A} \multimap \mathbf{C}}{by : \mathbf{C}} \multimap E}{< ax, by > : \mathbf{B} \otimes \mathbf{C}} \otimes I$$

Figure 2.6: Different labelled instances of the same formula

$$\dfrac{\boxed{b : \mathbf{A}} \quad a : \mathbf{A} \quad x : \mathbf{A} \multimap \mathbf{B}}{ax : \mathbf{B}} \multimap E \qquad \dfrac{\mathbf{A} \vdash \mathbf{A} \qquad \mathbf{B} \vdash \mathbf{B}}{\boxed{\mathbf{B}}, \mathbf{A}, \mathbf{A} \multimap \mathbf{B} \vdash \mathbf{B}}$$

Figure 2.7: Example of a *not allowed* discarding of a resource

computation, a factor that is very important for the good behaviour of algorithms, by using linear logic formulas to represent programs and memory cells [Lin92b]. This and other parallels that can be established with computation highlight the importance of linear logic's resource awareness.

2.2.3 Double Negation

Another very interesting feature of linear logic is incorporated in the linear negation connective, $(.)^{\perp}$, which is the most important connective of the whole logic according to Girard. With linear negation, *double negation* ($\mathbf{A} \equiv \mathbf{A}^{\perp\perp}$) holds, but the logic remains constructive. In classical logic, double negation ($\mathbf{A} \equiv \neg\neg\mathbf{A}$) also holds but that is seen as the cause for the logic not being constructive, since intuitionistic logic rejects double negation and is constructive.

Let us see why classical negation is not constructive. In classical logic sequent calculus we have the following rules for the negation connective:

$$\frac{\mathbf{\Gamma}, \mathbf{A} \vdash \mathbf{\Delta}}{\mathbf{\Gamma} \vdash \neg\mathbf{A}, \mathbf{\Delta}}\text{Right}\neg \quad \textit{and} \quad \frac{\mathbf{\Gamma} \vdash \mathbf{A}, \mathbf{\Delta}}{\mathbf{\Gamma}, \neg\mathbf{A} \vdash \mathbf{\Delta}}\text{Left}\neg$$

From the point of view of intuitionistic semantics, these rules allow one to prove that $\mathbf{A} \equiv \neg\neg\mathbf{A}$, i.e. that the sequent $\vdash (\neg\neg\mathbf{A} \Rightarrow \mathbf{A}) \wedge (\mathbf{A} \Rightarrow \neg\neg\mathbf{A})$ has a proof[3],

[3]In the proofs that follow we abbreviate Right $\neg$ to $\neg R$ and Left $\neg$ to $\neg L$.

as we can see below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I}{\vdash \neg\mathbf{A}, \mathbf{A}}\neg R}{\neg\neg\mathbf{A} \vdash \mathbf{A}}\neg L}{\vdash \neg\neg\mathbf{A} \Rightarrow \mathbf{A}}\Rightarrow R \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I}{\mathbf{A}, \neg\mathbf{A} \vdash}\neg L}{\mathbf{A} \vdash \neg\neg\mathbf{A}}\neg R}{\vdash \mathbf{A} \Rightarrow \neg\neg\mathbf{A}}\Rightarrow R}{\vdash (\neg\neg\mathbf{A} \Rightarrow \mathbf{A}) \wedge (\mathbf{A} \Rightarrow \neg\neg\mathbf{A})}\wedge R$$

In a similar way one can prove that $\mathbf{A} \equiv \mathbf{A}^{\perp\perp}$ in linear logic. Also, using the negation rules and the Contraction and Exchange structural rules, one can prove $\mathbf{A} \vee \neg\mathbf{A}$, i.e. that $\mathbf{A} \vee \neg\mathbf{A}$ holds without depending on any assumption:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I}{\vdash \neg\mathbf{A}, \mathbf{A}}\neg R}{\vdash \mathbf{A} \vee \neg\mathbf{A}, \mathbf{A}}\vee R}{\vdash \mathbf{A}, \mathbf{A} \vee \neg\mathbf{A}}\text{Exchange R}}{\vdash \mathbf{A} \vee \neg\mathbf{A}, \mathbf{A} \vee \neg\mathbf{A}}\vee R}{\vdash \mathbf{A} \vee \neg\mathbf{A}}\text{Contraction R}$$

And this is a very useful feature for certain proofs in classical logic.

Double negation is important in classical logic also because it allows *proof by absurd*. This kind of proof happens in the following way: in classical logic, $\neg\mathbf{A}$ is the same that $\mathbf{A} \Rightarrow \mathcal{F}$ (see Figure 2.8), where $\mathcal{F}$ is a formula which is always assigned to the truth-value 'false' by the truth assignment function. Since $\neg\neg\mathbf{A}$ is equivalent to $\mathbf{A}$, if you try to prove $\neg\mathbf{A}$ and find $\mathcal{F}$, then you have a proof of $\mathbf{A}$ (because of the implication introduction rule, see the sequence of rules in Figure 2.9). This is adequate for certain applications but not for others, because it is an indirect, non constructive way of proving $\mathbf{A}$.

$$A1 \quad \neg\mathbf{A} \equiv \mathbf{A} \Rightarrow \mathcal{F}$$
$$A2 \quad \mathbf{A} \equiv \neg\neg\mathbf{A}$$

Figure 2.8: Axioms for classical negation

$$\frac{\begin{array}{c}[\mathbf{A}]\\ \vdots \\ \mathbf{B}\end{array}}{\mathbf{A} \Rightarrow \mathbf{B}} \Rightarrow I \qquad \frac{\begin{array}{c}[\mathbf{A}]\\ \vdots \\ \mathcal{F}\end{array}}{\mathbf{A} \Rightarrow \mathcal{F}} \neg I_1 \qquad \frac{\begin{array}{c}[\mathbf{A}]\\ \vdots \\ \mathcal{F}\end{array}}{\neg\mathbf{A}} \neg I_2 \qquad \frac{\begin{array}{c}[\neg\mathbf{A}]\\ \vdots \\ \mathcal{F}\end{array}}{(\neg\mathbf{A}) \Rightarrow \mathcal{F}} \neg I_1 a \qquad \frac{\begin{array}{c}[\neg\mathbf{A}]\\ \vdots \\ \mathcal{F}\end{array}}{\neg\neg\mathbf{A}} \neg I_2 a \qquad \frac{\begin{array}{c}[\neg\mathbf{A}]\\ \vdots \\ \mathcal{F}\end{array}}{\mathbf{A}} \neg I_2$$

Figure 2.9: Proof by absurd in classical logic

2.3 Formulation of the System

Since Girard presented his system as a refinement of classical logic in [Gir87], here we are going to describe the necessary steps in order to obtain the linear logic sequent calculus formulation from classical logic sequent calculus. Because of this kind of presentation, in [Gir87] linear connectives were introduced as 'technical' transformations, not arising from semantical observations.

For each step we present also its justification, which is related to the intended features of the system presented in Section 2.2. We end up by showing all linear logic sequent calculus rules.

2.3.1 Steps in the Formulation

After having established the desired features of linear logic, by conceiving a notion of semantics of computation based on the so-called 'coherence spaces', Girard gave a formulation of his new 'resource-sensitive' logic. He divided the derivation of the system from the sequent calculus formulation of classical logic [Gen35] into four major steps:

1. Drop the *Contraction* and *Weakening* structural rules;

2. Introduce the distinction between *multiplicative* and *additive* connectives;

3. Introduce the *exponentials* (reintroducing Contraction and Weakening in a controlled form);

4. Introduce *linear negation.*

Dropping Structural Rules

According to Girard [Gir88], the structural rules are the most important of Gentzen's sequent calculus and determine the future behavior of the logical operations. Thus, in order to achieve the notion of formulas as resources we have to drop the Weakening and Contraction structural rules (from classical logic sequent calculus) because, in the context of a proof, the former allows one to *discard* a resource whilst the latter allows one to *duplicate* a resource. See the classical logic rules below where **C** is a formula which is duplicated in the Contraction rule and discarded in the

$$\dfrac{\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I \qquad \dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I \qquad \dfrac{}{\mathbf{B} \vdash \mathbf{B}}I}{\mathbf{A}, \mathbf{A} \Rightarrow \mathbf{B} \vdash \mathbf{B}} \Rightarrow L}{\boxed{\mathbf{A}}, \boxed{\mathbf{A}}, \mathbf{A} \Rightarrow (\mathbf{A} \Rightarrow \mathbf{B}) \vdash \mathbf{B}} \Rightarrow L}{\boxed{\mathbf{A}}, \mathbf{A} \Rightarrow (\mathbf{A} \Rightarrow \mathbf{B}) \vdash \mathbf{B}} CL$$

Figure 2.10: Duplication of a formula in classical logic

$$\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I \qquad \dfrac{\dfrac{}{\mathbf{B} \vdash \mathbf{B}}I}{\mathbf{C}, \mathbf{B} \vdash \mathbf{B}} WL}{\mathbf{A}, \boxed{\mathbf{C}}, \mathbf{A} \Rightarrow \mathbf{B} \vdash \mathbf{B}} \Rightarrow L$$

Figure 2.11: Discarding of a formula in classical logic

Weakening rule:

$$\textbf{Right Weakening} \quad \dfrac{\boldsymbol{\Gamma} \vdash \boldsymbol{\Delta}}{\boldsymbol{\Gamma} \vdash \mathbf{C}, \boldsymbol{\Delta}} \qquad \dfrac{\boldsymbol{\Gamma} \vdash \boldsymbol{\Delta}}{\boldsymbol{\Gamma}, \mathbf{C} \vdash \boldsymbol{\Delta}} \quad \textbf{Left Weakening}$$

$$\textbf{Right Contraction} \quad \dfrac{\boldsymbol{\Gamma} \vdash \mathbf{C}, \mathbf{C}, \boldsymbol{\Delta}}{\boldsymbol{\Gamma} \vdash \mathbf{C}, \boldsymbol{\Delta}} \qquad \dfrac{\boldsymbol{\Gamma}, \mathbf{C}, \mathbf{C} \vdash \boldsymbol{\Delta}}{\boldsymbol{\Gamma}, \mathbf{C} \vdash \boldsymbol{\Delta}} \quad \textbf{Left Contraction}$$

In classical logic, these rules are used, for example, to prove $\mathbf{A} \vee \neg\mathbf{A}$. The duplication (reuse) of a formula during a proof is illustrated in Figure 2.10 and the discarding of a formula is shown in Figure 2.11.

Linear logic is thus a *substructural* logic [Tro??], that is, a logic whose sequent calculus has *less* structural rules than classical logic, because it rejects two of the three structural rules of the sequent calculus originally presented in [Gen35].

Concerning structural rules, it is also interesting to notice that the Exchange rule

may become *unnecessary* in linear logic and therefore be excluded from its formulation (we are *not* considering this in the presentation of linear logic rules in Figure 2.16). This is done by reading the formulas in a sequent as a *multiset* (set with multiplicities) rather than as a sequence of formulas. In classical logic, a similar interpretation is also possible, only differing in the fact that there the formulas in a sequent are taken to form a *set* of formulas. The justification for this is that in both cases the order of appearance of the formulas in a sequent does not matter (because of the Exchange rule), but in linear logic the number of instances of a formula in a sequent does matter.

Multiplicative and Additive Connectives

The second step in deriving linear logic sequent rules is to differentiate between the two ways of formulating the rules for conjunction and disjunction. In classical logic, for example, it is possible to present the system with two different formulations for the right inference rule of the *and* connective, one additive and other multiplicative (see Figure 2.12). This is possible for each *and* and *or* rule. However, due to the structural rules, in classical logic these two types of formulation are *provably equivalent*. This means that the system with only multiplicative rules proves exactly the same sequents that the system with only additive rules [Sch94].

The lack of the structural rules in linear logic, however, makes it possible to regard each formulation as a different connective. Therefore, Girard presents his full system of linear logic with *two* conjunctions and *two* disjunctions. The rules for multiplicative conjunction ($\otimes$) and multiplicative disjunction ($\wp$) are formulated in a similar way to the second rule in Figure 2.12: the context ($\mathbf{\Gamma}'$, $\mathbf{\Gamma}''$ on the

$$\textbf{Additive } \wedge \quad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \qquad \frac{\Gamma' \vdash A, \Delta' \quad \Gamma'' \vdash B, \Delta''}{\Gamma', \Gamma'' \vdash A \wedge B, \Delta', \Delta''} \quad \textbf{Multiplicative}$$

Figure 2.12: Additive and multiplicative rules

lefthand side and Δ', Δ'' on the righthand side of the sequent) is not shared, but rather divided in the proof search[4]. The additive conjunction (&) and the additive disjunction ($\oplus$) rules, however, allow the sharing of contexts: in the first rule in Figure 2.12, a formula that appears in Γ or Δ (sequences of formulas) will be used at least twice along the derivation. Therefore, like other resource-sensitive logics (e.g. relevant logics), linear logic differentiates between these two possible formulations of the rules for conjunction and disjunction, considering them to be formulations of *distinct* connectives.

Exponentials

In order to recover the expressive power of intuitionistic logic, Girard presented the third step in the formulation of the system, namely the introduction of exponential connectives: ! (*Of course*) and ? (*Why not*). The exponentials allow one to express some propositions as *stable truths*. That is, the formulas *marked* by the exponentials can be *used* an unlimited number of times. In proof-theoretic terms, this means that Contraction and Weakening can be used in those formulas (see exponential rules in Figure 2.16). As a result, it is indeed possible to give a trans-

[4]These connectives are *more linear*, because even in the course of a derivation a formula has to be used exactly once. That is why the meaning of a sequent $\Gamma \vdash \Delta$ is established as $\otimes\Gamma \multimap \wp\Delta$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}}I \quad \dfrac{}{\mathbf{B} \vdash \mathbf{B}}I}{\mathbf{A}, \mathbf{A} \multimap \mathbf{B} \vdash \mathbf{B}} \multimap L}{\mathbf{A}, \mathbf{A}, \mathbf{A} \multimap \mathbf{B} \vdash \mathbf{A} \otimes \mathbf{B}} \otimes R}{\mathbf{A}, !\mathbf{A}, \mathbf{A} \multimap \mathbf{B} \vdash \mathbf{A} \otimes \mathbf{B}} !D}{!\mathbf{A}, !\mathbf{A}, \mathbf{A} \multimap \mathbf{B} \vdash \mathbf{A} \otimes \mathbf{B}} !D}{!\mathbf{A}, \mathbf{A} \multimap \mathbf{B} \vdash \mathbf{A} \otimes \mathbf{B}} !C$$

Figure 2.13: The use of exponentials

lation of intuitionistic logic sequents into linear logic sequents where the latter are provable if and only if the former are provable (i.e. the translation is correct and complete).

In Figure 2.13 we can see an example where the exponential mark (!) on a formula **A** in the lefthand side of the sequent permits one to use the formula twice: first to produce **B** and then to produce $\mathbf{A} \otimes \mathbf{B}$. One can also notice that in order to use a formula that is marked by an exponential, it is necessary first to *remove* the exponential mark. This is done by using a new structural rule, Dereliction, specially created for this purpose. Although the expressive power of intuitionistic and classical logic is somehow restored, one can see that this introduces a little confusion in the resulting proof system (i.e. there are too many rules).

Linear Negation

The fourth step in the derivation of linear logic is the introduction of linear negation $((.)^{\perp}$, also called 'perp'), which is, according to Girard, the most important

$$\mathbf{A} \equiv \mathbf{A}^{\perp\perp} \qquad (\mathbf{A} \otimes \mathbf{B})^{\perp} \equiv \mathbf{A}^{\perp} \wp \mathbf{B}^{\perp} \quad (\mathbf{A} \wp \mathbf{B})^{\perp} \equiv \mathbf{A}^{\perp} \otimes \mathbf{B}^{\perp}$$

$$(\mathbf{A} \& \mathbf{B})^{\perp} \equiv \mathbf{A}^{\perp} \oplus \mathbf{B}^{\perp} \quad (\mathbf{A} \oplus \mathbf{B})^{\perp} \equiv \mathbf{A}^{\perp} \& \mathbf{B}^{\perp} \qquad (!\mathbf{A})^{\perp} \equiv ?\mathbf{A}^{\perp}$$

$$(?\mathbf{A})^{\perp} \equiv !\mathbf{A}^{\perp} \qquad (\forall \alpha.\mathbf{A})^{\perp} \equiv \exists \alpha.\mathbf{A}^{\perp} \qquad (\exists \alpha.\mathbf{A})^{\perp} \equiv \forall \alpha.\mathbf{A}^{\perp}$$

Figure 2.14: DeMorgan derived equalities

$$\dfrac{\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}} I \qquad \dfrac{}{\mathbf{B} \vdash \mathbf{B}} I}{\mathbf{A}, \mathbf{B} \vdash \mathbf{A} \otimes \mathbf{B}} \otimes R}{\mathbf{A}, \mathbf{B}, (\mathbf{A} \otimes \mathbf{B})^{\perp} \vdash} \perp L \qquad \dfrac{\dfrac{\dfrac{}{\mathbf{A} \vdash \mathbf{A}} I}{\mathbf{A}, \mathbf{A}^{\perp} \vdash} \perp L \qquad \dfrac{\dfrac{}{\mathbf{B} \vdash \mathbf{B}} I}{\mathbf{B}, \mathbf{B}^{\perp} \vdash \mathbf{B}} \perp L}{\mathbf{A}, \mathbf{B}, \mathbf{A}^{\perp} \wp \mathbf{B}^{\perp} \vdash} \wp L$$

Figure 2.15: Example of the interchange of dual formulas

connective of linear logic. Linear negation, as we have already said, allows 'double negation' without losing constructivity. It is introduced in the sequent calculus formulation of linear logic by the negation rules in Figure 2.16 and it appears in the set of derived equations that express DeMorgan-like *dualities* (Figure 2.14). One interesting property of linear negation is that, in a proof, dual formulas can be used interchangeably without modifying the result (see an example in Figure 2.15, where the dual formulas are $(\mathbf{A} \otimes \mathbf{B})^{\perp}$ and $\mathbf{A}^{\perp} \wp \mathbf{B}^{\perp}$).

Having presented the four steps that allow one to 'translate' classical logic sequent calculus into linear logic sequent calculus, we are able to present the full system of linear logic. This system, which is composed of the rules in Figure 2.16 is called

Identity	$\dfrac{}{\mathbf{A} \vdash \mathbf{A}}$	$\dfrac{\Gamma' \vdash \mathbf{A}, \Delta' \quad \Gamma'', \mathbf{A} \vdash \Delta''}{\Gamma', \Gamma'' \vdash \Delta', \Delta''}$	**Cut**
Left Exchange	$\dfrac{\Gamma, \mathbf{B}, \mathbf{A} \vdash \Delta}{\Gamma, \mathbf{A}, \mathbf{B} \vdash \Delta}$	$\dfrac{\Gamma \vdash \mathbf{B}, \mathbf{A}, \Delta}{\Gamma \vdash \mathbf{A}, \mathbf{B}, \Delta}$	**Right Exchange**
Left $\otimes$	$\dfrac{\Gamma, \mathbf{A}, \mathbf{B} \vdash \Delta}{\Gamma, \mathbf{A} \otimes \mathbf{B} \vdash \Delta}$	$\dfrac{\Gamma' \vdash \mathbf{A}, \Delta' \quad \Gamma'' \vdash \mathbf{B}, \Delta''}{\Gamma', \Gamma'' \vdash \mathbf{A} \otimes \mathbf{B}, \Delta', \Delta''}$	**Right** $\otimes$
Left $\wp$	$\dfrac{\Gamma', \mathbf{A} \vdash \Delta' \quad \Gamma'', \mathbf{B} \vdash \Delta''}{\Gamma', \Gamma'', \mathbf{A} \wp \mathbf{B} \vdash \Delta', \Delta''}$	$\dfrac{\Gamma \vdash \mathbf{A}, \mathbf{B}, \Delta}{\Gamma \vdash \mathbf{A} \wp \mathbf{B}, \Delta}$	**Right** $\wp$
Left $\&_1$	$\dfrac{\Gamma, \mathbf{A} \vdash \Delta}{\Gamma, \mathbf{A} \& \mathbf{B} \vdash \Delta}$	$\dfrac{\Gamma, \mathbf{B} \vdash \Delta}{\Gamma, \mathbf{A} \& \mathbf{B} \vdash \Delta}$	**Left** $\&_2$
Right $\&$	$\dfrac{\Gamma \vdash \mathbf{A}, \Delta \quad \Gamma \vdash \mathbf{B}, \Delta}{\Gamma \vdash \mathbf{A} \& \mathbf{B}, \Delta}$	$\dfrac{\Gamma, \mathbf{A} \vdash \Delta \quad \Gamma, \mathbf{B} \vdash \Delta}{\Gamma, \mathbf{A} \oplus \mathbf{B} \vdash \Delta}$	**Left** $\oplus$
Right $\oplus_1$	$\dfrac{\Gamma \vdash \mathbf{A}, \Delta}{\Gamma \vdash \mathbf{A} \oplus \mathbf{B}, \Delta}$	$\dfrac{\Gamma \vdash \mathbf{B}, \Delta}{\Gamma \vdash \mathbf{A} \oplus \mathbf{B}, \Delta}$	**Right** $\oplus_2$
Left $^\perp$	$\dfrac{\Gamma \vdash \mathbf{A}, \Delta}{\Gamma, \mathbf{A}^\perp \vdash \Delta}$	$\dfrac{\Gamma, \mathbf{A} \vdash \Delta}{\Gamma \vdash \mathbf{A}^\perp, \Delta}$	**Right** $^\perp$
! Contraction	$\dfrac{\Gamma, \mathbf{!A}, \mathbf{!A} \vdash \Delta}{\Gamma, \mathbf{!A}, \vdash \Delta}$	$\dfrac{\Gamma \vdash \Delta}{\Gamma, \mathbf{!A} \vdash \Delta}$	**! Weakening**
! Dereliction	$\dfrac{\Gamma, \mathbf{A} \vdash \Delta}{\Gamma, \mathbf{!A} \vdash \Delta}$	$\dfrac{!\Gamma \vdash \mathbf{A}, ?\Delta}{!\Gamma \vdash \mathbf{!A}, ?\Delta}$	**Of Course!**
? Contraction	$\dfrac{\Gamma \vdash \mathbf{?A}, \mathbf{?A}, \Delta}{\Gamma \vdash \mathbf{?A}, \Delta}$	$\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash \mathbf{?A}, \Delta}$	**? Weakening**
? Dereliction	$\dfrac{\Gamma \vdash \mathbf{A}, \Delta}{\Gamma \vdash \mathbf{?A}, \Delta}$	$\dfrac{!\Gamma, \mathbf{A} \vdash ?\Delta}{!\Gamma, \mathbf{?A} \vdash ?\Delta}$	**Why not?**
$\multimap$ definition	$(\mathbf{A} \multimap \mathbf{B}) \equiv \mathbf{A}^\perp \wp \mathbf{B}$		

Figure 2.16: Rules for the connectives

MODIFICATION	JUSTIFICATION
Rejection of Contraction and Weakening structural rules	Not allow the discarding or duplication of formulas along a proof
Distinction between multiplicative and additive formulations of connectives	The lack of structural rules makes it possible to regard each formulation as a different connective
Introduction of exponentials	Recover the expressive power of intuitionistic logic
Introduction of linear negation	To have double negation without losing constructivity

Table 2.2: Summary of the modifications

(classical) propositional linear logic, or plainly linear logic. If we add to this system first-order quantifier rules, we will have (classical) first-order linear logic. We can also obtain *intuitionistic* versions of these systems by restricting the righthand side of sequents to have *at most* one formula. Next we discuss these and other possible modifications to linear logic.

2.4 Linear Logic Fragments

Linear logic *fragments* are subsystems of linear logic, i.e. systems with less rules and/or connectives than the full linear logic system (propositional linear logic). The idea of having fragments of logical systems is not new in logic; implicative intuitionistic logic, for example, is intuitionistic logic with only one connective:

intuitionistic implication. The study of fragments gains importance in linear logic because of its large number of connectives, a feature that allows the formation of several different linear subsystems.

There are two main (and interrelated) reasons for the relevance of the study of linear logic fragments. The first reason is that some of these fragments are expressive enough to represent faithfully certain computer science applications. Therefore, there is no need to use the entire system, which is more complicated, to represent these applications. The second reason is related to the different complexity measures of the several fragments. The computational complexity of linear logic fragments has been studied [Lin92a, Lin92b, Kan92, Laf94] and smaller fragments were (rather obviously) found to be less computationally complex than bigger fragments. The full system of propositional linear logic, even without the addition of quantifiers, was found to be undecidable. Therefore, everything that can be expressed in a smaller fragment will usually be expressed in a (hopefully) decidable fragment.

Besides these two reasons, we identify an important side effect of these studies: by studying the computational complexity of linear logic fragments one gains more *insight* about the role of each connective, or group of connectives, in the full proof system. There are several such results, such as the finding that the exponentials are responsible for the undecidability of the full system. In this way, the study of linear logic fragments and its properties can provide important information as to how to represent computer science applications.

2.4.1 Formation of the Fragments

The works in the literature that study linear logic fragments do not usually give more thorough explanations about the formation (and significance) of these fragments. This formation is, in fact, relatively obvious. That is, in Girard's presentations of linear logic (e.g. [Gir95b]), he establishes very clearly some *groups* of connectives and rules (such as multiplicative and additive connectives). Then, fragments are formed by including or excluding these groups from the full system, and one names these fragments by forming acronyms with the first letters of the groups' names.

Since our intention here is also to discuss the role of each connective or group of connectives in the full system, we present a simple framework to describe this formation of fragments. The framework consists first of having groups of rules, equations and axioms, which are called 'building blocks'. We form fragments by adjoining one or more building blocks. Secondly, we can also have *restrictions* that when applied to fragments produce different fragments.

Now we present some building blocks (there are others that are not shown here) for linear logic. We present them in a 'somehow' natural order and associate each block to one or more letters. These letters are going to be later used in order to form the acronyms that will name the fragments. The building blocks are derived from rather obvious observations about the subsystems of linear logic presented in the literature. The main building blocks in this framework are:

- Multiplicative connectives (M): It is the most important block, the one that appears in most fragments. It contains the inference rules for the multiplica-

Structural rules	**Rules for connectives**	**Equations**
Identity	**Left** $\otimes$	$\mathbf{A} \equiv \mathbf{A}^{\perp\perp}$
Cut	**Right** $\otimes$	$(\mathbf{A} \otimes \mathbf{B})^{\perp} \equiv \mathbf{A}^{\perp} \wp \mathbf{B}^{\perp}$
Left Exchange	**Left** $\wp$	$(\mathbf{A} \wp \mathbf{B})^{\perp} \equiv \mathbf{A}^{\perp} \otimes \mathbf{B}^{\perp}$
Right Exchange	**Right** $\wp$	$(\mathbf{A} \multimap \mathbf{B}) \equiv \mathbf{A}^{\perp} \wp \mathbf{B}$

Table 2.3: The multiplicative building block

tive connectives plus the Exchange structural rules, the Identity rule and the Cut rule, as well as the rules for linear negation[5]. The fragment that contains only this block (see Figure 2.3) is called MLL (Multiplicative Linear Logic).

This block is also important because multiplicative connectives are the meaning of commas in a sequent, i.e. $\mathbf{\Gamma} \vdash \mathbf{\Delta}$ is a provable sequent if and only if $\vdash \otimes\mathbf{\Gamma} \multimap \wp\mathbf{\Delta}$ is also provable[6]. It is important to notice that $\multimap$ (linear implication) is a defined connective, derived from multiplicative conjunction and linear negation ($\mathbf{A} \multimap \mathbf{B} \equiv \mathbf{A}^{\perp} \wp \mathbf{B}$), therefore it is included in the multiplicative block.

- Additive connectives (A): This bulding block consists of the rules concerning the additive connectives (& and $\oplus$) as well as Exchange structural rules, the Identity rule and the Cut rule. It is possible to have a purely additive fragment (ALL-Additive Linear Logic), but this is a very restricted frag-

[5]We have not seen any linear logic fragment without negation in the literature.

[6]If $\mathbf{\Gamma} = \mathbf{A}, \mathbf{B}, \mathbf{C}$ then $\otimes\mathbf{\Gamma}$ is $\mathbf{A} \otimes \mathbf{B} \otimes \mathbf{C}$.

ment (see [Mar96]). A well-known fragment that contains this block is the one that joins it to the multiplicative block: MALL (Multiplicative Additive Linear Logic). For some applications in computer science, the additive connectives are essential, since they allow the representation of certain features of computation, such as the idea of 'choice' [Lin92a].

- Exponentials (E): As we have already said, the exponential connectives allow linear logic to restore the expressive power of intuitionistic logic. Therefore, this block, which contains the rules involving the connectives ! and ?, is fundamental for several applications of linear logic to computation. The representation of Petri nets [Ale94], for example, uses exponentials to represent transitions. The smallest fragment including this block is MELL (Multiplicative Exponential Linear Logic). The exponentials are also seen as responsible for the undecidability of the full system of linear logic, since MLL and MALL are decidable.

- First-order quantifiers (1): The next 'natural' building block is the one that contains the rules for the first-order quantifiers ($\forall$ and $\exists$). According to Girard [Gir89], the quantifier rules (see Figure 2.17) are not different from those of classical logic. This is understandable since, in sequent calculus, terms are not first-class citizens [dG92]; it would be impossible to impose restrictions on the use of terms in the sequent calculus in the same manner that restrictions are imposed on the use of formulas. MLL1 (First-order MLL) is the first fragment including quantifiers, but it is not much used. In the literature, first-order quantifiers usually appear forming first-order linear logic (called MAELL1 or plainly first-order LL).

$$\textbf{Left } \exists \quad \frac{\Gamma \vdash \mathbf{A}[\mathbf{t}], \Delta}{\Gamma \vdash \exists \mathbf{x}\ \mathbf{A}[\mathbf{x}], \Delta}$$

$$\textbf{Right } \exists \quad \frac{\Gamma, \mathbf{A}[\mathbf{t}] \vdash \Delta}{\Gamma, \exists \mathbf{x}\ \mathbf{A}[\mathbf{x}] \vdash \Delta} (\mathbf{x} \textit{ is not free in } \Gamma \textit{ and } \Delta)$$

$$\textbf{Left } \forall \quad \frac{\Gamma \vdash \mathbf{A}[\mathbf{t}], \Delta}{\Gamma \vdash \forall \mathbf{x}\ \mathbf{A}[\mathbf{x}], \Delta} (\mathbf{x} \textit{ is not free in } \Gamma \textit{ and } \Delta)$$

$$\textbf{Right } \forall \quad \frac{\Gamma, \mathbf{A}[\mathbf{t}] \vdash \Delta}{\Gamma, \forall \mathbf{x}\ \mathbf{A}[\mathbf{x}] \vdash \Delta}$$

Figure 2.17: Rules for first-order quantifiers

$$\textbf{Left } \exists \quad \frac{\Gamma \vdash \mathbf{A}[\mathbf{V}], \Delta}{\Gamma \vdash \exists \mathbf{X}\ \mathbf{A}[\mathbf{X}], \Delta}$$

$$\textbf{Right } \exists \quad \frac{\Gamma, \mathbf{A}[\mathbf{V}] \vdash \Delta}{\Gamma, \exists \mathbf{X}\ \mathbf{A}[\mathbf{X}] \vdash \Delta} (\mathbf{X} \textit{ is not free in } \Gamma \textit{ and } \Delta)$$

$$\textbf{Left } \forall \quad \frac{\Gamma \vdash \mathbf{A}[\mathbf{V}], \Delta}{\Gamma \vdash \forall \mathbf{X}\ \mathbf{A}[\mathbf{X}], \Delta} (\mathbf{X} \textit{ is not free in } \Gamma \textit{ and } \Delta)$$

$$\textbf{Right } \forall \quad \frac{\Gamma, \mathbf{A}[\mathbf{V}] \vdash \Delta}{\Gamma, \forall \mathbf{X}\ \mathbf{A}[\mathbf{X}] \vdash \Delta}$$

Figure 2.18: Rules for second-order quantifiers

- Second-order quantifiers (2): Second-order quantifiers are commonly used in order to represent polymorphism, as in Girard's System F [Gir88]. These quantifiers do not quantify over terms, but over types. That is why the are called *second-order* quantifiers. This block consists of the rules for these second-order quantifiers, represented in Figure 2.18 by the same symbols that represent first-order quantifiers. MLL2 (Second-order MLL) is the smallest possible fragment including second-order quantifiers and it was proved undecidable in [Laf88].

- Bounded exponentials (B): The bounded exponentials have appeared in [GSS92]. In this system, exponentially marked formulas are used only a *limited* number of times. There are two formulations there: the simpler one, which is included in MALL and is presented only to illustrate the idea, establishes that $!_{\mathbf{n}}\mathbf{A} = \underbrace{\mathbf{A} \otimes \ldots \otimes \mathbf{A}}_{n \textbf{ times}}$. There is also a more complicated formulation with a similar meaning. This block consists of this more complicated set of rules for the bounded exponentials. The idea behind this definition is to try to develop a system that can represent polynomial time computations. According to Girard, this system, BLL—Bounded Linear Logic, was not very successful. However, this development led to other systems with better results: ELL (Elementary Linear Logic) and LLL (Light Linear Logic) [Gir95b].

Besides these building blocks consisting of connectives and rules, we can also form new fragments by applying *restrictions* to existing fragments. There may be several different types of restrictions and here we present only three of them:

- Intuitionistic versions (I): In Gentzen's sequent calculus, it is possible to

present a formulation of intuitionistic logic by restricting classical logic formulation. There are several possibilities for doing this; one of them, the restriction of having at most one formula in the righthand side of sequents, can also be used in linear logic. IMLL (Intuitionistic MLL), for example, is the system with the rules of the multiplicative block whose proofs obbey this restriction; another example is FILL (Full Intuitionistic Linear Logic), a variant of (multiplicative and exponential-free) Linear Logic introduced by Hyland and de Paiva (see [BdP96]).

The intuitionistic versions of linear logic are important because, among other things, they allow a representation of *linear* functional programming. However, they have some problems concerning the dualities; for example, $\wp$ right rule cannot be used because it demands that two formulas appear in the righthand side of the sequent.

- Non-commutative versions (N): According to Girard [Gir95b], it is fairly natural to think of non-commutative versions of linear logic. This happens because of two reasons. First, since linear logic appears as a modification of classical logic that removes two structural rules, the next natural step is to ask whether one can also remove the remaining structural rule: the Exchange rule. With this removal, the sequents are not viewed as sequences, like in classical logic [Gen35], neither as multisets, like in (commutative) linear logic, but rather as lists or circular lists [Lin92a].

  Second, for some applications in computer science the position of each data item in a data structure matters. Therefore, when these applications are represented in linear logic, the position of each formula in a sequent is rele-

vant for the proof. An example is the representation of data types, such as stacks. However, it is difficult to decide amongst the several different options for non-commutative versions of linear logic. For instance, some versions consider that a sequent is a list of formulas; others consider a sequent as a *circular* list of formulas. Another modification may occur in the order of the formulas in a negation equation (instead of $(\mathbf{A} \oplus \mathbf{B})^{\perp} \equiv \mathbf{A}^{\perp} \& \mathbf{B}^{\perp}$ we would have $(\mathbf{A} \oplus \mathbf{B})^{\perp} \equiv \mathbf{B}^{\perp} \& \mathbf{A}^{\perp}$). Each application may need different solutions and therefore the question of which is the better non-commutative version is still open.

A possible NCMLL (Non-Commutative MLL) would be an MLL where the only modification were the removal of the Exchange rule. This system is too restricted [Lin92a] and therefore a different non-commutative version may be necessary. For example, a version where the Exchange rule is substituted by a Circular rule, that allows the rotation of the formulas in a sequent. Notwithstanding this indefinition, non-commutative linear logic has a big expressive power [Gir89, Lin92a] and has found a lot of applications in theoretical computer science, among other areas.

- Constant-only (Co): Amongst the restrictions presented here this is the more uncommon. The idea is to consider a system where the only propositions are linear logic constants (or units). Each additive or multiplicative connective has its constant, whose main characteristic is that, for any formula $\mathbf{A}$, given the connective $\circ$ and its unit $\mathbf{U}$, $\mathbf{A} \circ \mathbf{U} \equiv \mathbf{A}$. $\mathbf{1}$ is the constant for connective $\otimes$, $\perp$ is the unit for connective $\wp$, $\top$ is the unit for connective $\&$ and $\mathbf{0}$ is the unit for connective $\oplus$. The most interesting result that was obtained

by studying these fragments [Kan92, Lin92a] was the finding that even these severely restricted fragments had a big expressive power and, consequently, a high computational complexity. CoMLL (Constant-only MLL), for example, which is the simplest possible constant-only fragment, is NP-complete.

In Figure 2.4 we see some of the fragments that we can construct by adjoining the building blocks presented above and by applying restrictions. The acronyms giving name to the fragments are formed by using the letters that accompany the blocks and restrictions presented. Most of these linear logic fragments have found some application in computer science. For example, MELL is used to represent Petri Nets [Ale94] (although not with completeness), Intuitionistic Linear Logic was given a term assignment in the spirit of the Curry-Howard isomorphism which showed to be a refinement of the λ-calculus (a linear λ-calculus where some memory management operations like `copy, read, discard` and `store` are *explicit* in the terms of the calculus [Abr93, Lin92a, Ale94]). Of course some combinations of building blocks and restrictions are quite rare in practice, like CoMELL. Notwithstanding, most of these combinations enjoy a very important property, cut-elimination.

2.4.2 Right-only Sequents

Finally, a very important modification to the presentation of linear logic fragments, which was not presented before because it does not change the expressive power of the systems, is the consideration of right-only sequents. The idea is the following: for any fragment it is possible (due to 'De Morgan'-dualities) to formulate the

Acronym	**Complete Name**
MLL	Multiplicative Linear Logic
MALL	Muliplicative Additive LL
MELL	Multiplicative Exponencial LL
MAELL or LL	(Classical) Linear Logic
IMAELL or ILL	Intutionistic Linear Logic
ILL2	Second order ILL
CoMLL	Constant-only MLL

Table 2.4: Fragments of linear logic

sequent calculus with rules where the formulas appear only in the righthand side of the sequent (right-sided sequents). This is only a matter of economy since for each connective now it suffices to have only one rule. For example, the cut rule would be transformed in the following way:

$$\text{Instead of} \quad \frac{\mathbf{\Gamma} \vdash \mathbf{A}, \mathbf{\Delta} \quad \mathbf{\Gamma}', \mathbf{A} \vdash \mathbf{\Delta}'}{\mathbf{\Gamma}, \mathbf{\Gamma}' \vdash \mathbf{\Delta}, \mathbf{\Delta}'} \quad \text{one can use} \quad \frac{\vdash \mathbf{A}, \mathbf{\Gamma}^{\perp}, \mathbf{\Delta} \quad \vdash \mathbf{A}^{\perp}, \mathbf{\Gamma}'^{\perp}, \mathbf{\Delta}'}{\vdash \mathbf{\Gamma}^{\perp}, \mathbf{\Gamma}'^{\perp}, \mathbf{\Delta}, \mathbf{\Delta}'}$$

where $\mathbf{\Gamma}^{\perp}$ is a multiset of the negation of all formulas in $\mathbf{\Gamma}$. For example, if $\mathbf{\Gamma} = \mathbf{A}, \mathbf{B}, \mathbf{C} \otimes \mathbf{B}$, then $\mathbf{\Gamma}^{\perp} = \mathbf{A}^{\perp}, \mathbf{B}^{\perp}, (\mathbf{C} \otimes \mathbf{B})^{\perp}$.

We can simplify the presentation of linear logic fragments by using *right-sided* sequents, and this facilitates the application of fragments to computer science. This application is also made easier by two other facts. The first one is that the fragments allow a flexibility in the choice of representation of these computer science aplications—i.e. it is not always necessary to use the full system. Second,

the use of fragments usually reduces the complexity of the system used to represent a given application, since the smaller the fragment, the less computational complex it is.

However, the existence of fragments highlights two problems of linear logic. In the first place, there are so many fragments only because linear logic has so many (maybe *too* many) different connectives. In the second place, smaller fragments are used because the full system is undecidable, i.e. it is *too* much computationally complex. And even these smaller fragments are very much complex (the minimum fragment, CoMLL, is NP-complete!). These two facts are relevant for computer science and can, therefore, prevent the use of linear logic as a serious logical tool for the representation of applications in this area.

2.5 Explanation of the Connectives

The intention here is to explain the meaning of linear logic connectives. We shall try to clarify this meaning by giving informal explanations of the connectives, as well as by presenting computational interpretations of and proof theoretic information on the connectives. The informal explanations are based on the use of the connectives along proofs in linear logic and are taken from [Gir89, Gir95b, Abr93, Laf88, Sch94]. The computational interpretation of some connectives were presented in works such as [Lin92b, Abr93, BS94] and the proof theoretic information was found in [Gir87, Gir95b, Tro??, Laf88]. This kind of explanation is necessary to introduce linear logic to computer science researchers, even for those who are acquainted with classical and intuitionistic logic or proof theory, since linear logic was first pre-

sented as a sequent calculus modification of classical logic (that is, the connectives appeared as 'technical' transformations, not arising from semantical observations).

These several levels of explanation are necessary because of one important reason: it is not easy to grasp the meaning of linear connectives only by looking at their sequent calculus rules. The fact is that there are many *new* connectives in linear logic, and some of them are really different from existing classical and intuitionistic connectives. Besides that, linear connectives carry more information (resource use information, for example) than classical connectives. Therefore, we have to present additional explanations for the connectives in the full linear logic system.

The explanations that we are going to present concern three features of linear connectives:

- **Resource use:** When explaining the resource use aspect of a connective, one is interested in how the use of a composite formula, whose main connective is being analysed, affects the use of its subformula(s)[7]. For example, in linear logic if the formula $\mathbf{A} \otimes \mathbf{B}$ is used, this means that the formulas **A** and **B** are used (exactly once). However, if **A**&**B** is used, only one of the subformulas, **A** *or* **B**, is used (exactly once). This aspect is very important because linear logic is a resource aware logic, where formulas are seen as resources.

- **Choice:** The choice aspect is related to the possibility of choosing the first or the second immediate subformula of a composite formula along a proof. This is better explained with an example. In classical logic, if one wants to

[7]Every composite formula has two immediate subformulas, except exponentiated and quantified formulas, which have only one.

prove $\mathbf{A} \vee \mathbf{B}$ he can choose to prove $\mathbf{A}$ or to prove $\mathbf{B}$. However, if he wants to prove $\mathbf{A} \wedge \mathbf{B}$ he *must* prove $\mathbf{A}$ and $\mathbf{B}$. That is, in the first case he has a choice but not in the second case. To explain the possibilities of choice when proving a formula whose main connective is a given linear logic connective is a way to explain this connective.

- **Duality:** It is important to know the connectives which are duals in linear logic. This information may be used in order to formulate and simplify the representation of applications. A connective $\circ$ is the dual of other connective $\bullet$ if and only if $(\mathbf{A} \circ \mathbf{B})^{\perp} \equiv \mathbf{A}^{\perp} \bullet \mathbf{B}^{\perp}$. The dualities in linear logic are rather obvious to find given the way the rules are formulated, except maybe the duality between the exponentials $((!\mathbf{A})^{\perp} \equiv ?\mathbf{A}^{\perp})$.

Having these three features in mind, we discuss the meaning of linear connectives by presenting Girard's explanations and other information about each connective. These explanations must be linked to the inference rules presented before.

- **Linear implication ($\multimap$):** In classical or intuitionistic logic, the implication connective allows one to express the following situation:

  If $\mathbf{A}$ *and* $\mathbf{A} \Rightarrow \mathbf{B}$*, then* $\mathbf{B}$*, but* $\mathbf{A}$ *still holds.*

  According to Girard, this is perfect in mathematics, but wrong in real life. Therefore, linear logic presents a *causal* implication, where

  If $\mathbf{A}$ *and* $\mathbf{A} \multimap \mathbf{B}$*, then* $\mathbf{B}$*, but* $\mathbf{A}$ *does not hold anymore.*

Intuitively, linear implication expresses that one uses (some) resources in order to produce (some) results. That is, if $\mathbf{A} \multimap \mathbf{B}$, by using $\mathbf{A}$ one can produce $\mathbf{B}$.

The comparison of the computational interpretation of linear implication to the computational interpretation of intuitionistic implication also helps to understand the meaning of this connective. In linear logic, implication is represented computationally by a (mathematical) function, just as intuitionistic implication, but it is a *linear* function [Lin92b, Abr93], i.e. a function that uses its arguments exactly once. For example, $f(x) = x + x$ and $f(x, y) = x$ are not linear functions (although they are 'intuitionistic' functions), but $f(x, y) = x + y$ is, because it uses each argument exactly once.

- **Times ($\otimes$):** Times is a conjunction that carries more information than classical conjunction because, due to the resource awareness of linear logic, if $(\mathbf{A} \otimes \mathbf{B}) \multimap \mathbf{C}$ then one must use $\mathbf{A}$ and $\mathbf{B}$ exactly once in order to produce $\mathbf{C}$. Two resources are grouped by this conjunction. Therefore, $\mathbf{B}$ is different from $\mathbf{B} \otimes \mathbf{B}$, since $\mathbf{B}$ means to have one copy of resource $\mathbf{B}$ whilst $\mathbf{B} \otimes \mathbf{B}$ means to have two copies of the same resource. Computationally, it is a kind of pair where both members have to be used once along the computation.

- **With (&):** This conjunction differs from the previous one in the resource use information. Here, in order to produce $\mathbf{C}$ from $(\mathbf{A}\&\mathbf{B}) \multimap \mathbf{C}$ one does not have to use $\mathbf{A}$ and $\mathbf{B}$. Rather, one must use one of them, but he can choose which one. The chosen formula, however, must be used exactly once. This conjunction has a disjunctive flavour but it is surely a conjunction, according to Girard, because $(\mathbf{A}\&\mathbf{B}) \multimap \mathbf{A}$ and $(\mathbf{A}\&\mathbf{B}) \multimap \mathbf{B}$ are valid formulas.

Since one cannot use both conjuncts at the same time (see & left rules in Figure 2.16, where either the **A** or the **B** is discarded when you use a formula where & is the main connective), this conjunction is computationally represented by a pair where exactly one of the members has to be used. That is, one of the members of the pair must be chosen by the user of the data at the time of computation.

The following extract from [Gir95b] is a good illustration of the difference between the two linear conjunctions:

> (...) consider **A**, **B**, **C**:
>
> **A**: to spend \$1,
>
> **B**: to get a pack of Camels,
>
> **C**: to get a pack of Marlboro.
>
> An action of type **A** will be a way of taking \$1 out of one's pocket (there may be several actions of this type since we own several notes). Similarly, there are several packs of Camels at the dealer's, hence there are several actions of type **B**. An action of type $\mathbf{A} \multimap \mathbf{B}$ is a way of replacing any specific dollar by a specific pack of Camels.
>
> Now, given an action of type $\mathbf{A} \multimap \mathbf{B}$ and an action of type $\mathbf{A} \multimap \mathbf{C}$, there will be no way of forming an action of type $\mathbf{A} \multimap \mathbf{B} \otimes \mathbf{C}$, since for \$1 you will never get what costs \$2 (there will be an action of type $\mathbf{A} \otimes \mathbf{A} \multimap \mathbf{B} \otimes \mathbf{C}$, namely getting two packs for \$2). However, there will be an action of type $\mathbf{A} \multimap \mathbf{B} \& \mathbf{C}$, namely the superimposition of both actions. In order to perform this action,

> we have first to choose which among the two possible actions we want to perform, and then to do the one selected. This is an exact analogue of the computer instructions **if ... then ... else ...**: in this familiar case, the parts **then ...** and **else ...** are available, but only one of them will be done. (...)

- **Par ($\wp$):** Par is probably the most confuse connective of linear logic. Nobody gives a better explanation than saying that $\mathbf{A}^{\perp} \wp \mathbf{B} \equiv \mathbf{A} \multimap \mathbf{B}$. It seems to be an open question to give a better intuitive or computational explanation. One could ask, for example, what is means an action $\mathbf{A} \multimap \mathbf{B} \wp \mathbf{C}$ in the context of the quotation above from [Gir95b], since for the two conjunctions and the other disjunction (see below) this question was answered. What would be produced given an action of type $\mathbf{A}$?

- **Plus ($\oplus$):** Plus has a clearer meaning than Par since it is very similar to intuitionistic disjunction. Here, $\mathbf{A} \multimap \mathbf{A} \oplus \mathbf{B}$ and $\mathbf{B} \multimap \mathbf{A} \oplus \mathbf{B}$, therefore, if you have $\mathbf{A} \oplus \mathbf{B}$, then you have $\mathbf{A}$ or $\mathbf{B}$, but you do not know which. This is clearly expressed in the rule

$$\frac{\mathbf{A} \vdash \mathbf{\Delta} \quad \mathbf{B} \vdash \mathbf{\Delta}}{\mathbf{A} \oplus \mathbf{B} \vdash \mathbf{\Delta}}$$

  where to prove that you have $\mathbf{\Delta}$ from $\mathbf{A} \oplus \mathbf{B}$ you must first prove that you have $\mathbf{\Delta}$ from $\mathbf{A}$ and from $\mathbf{B}$ independently. This happens because you do not know which one you have when you are going to use $\mathbf{A} \oplus \mathbf{B}$ (exactly the same that you have in the natural deduction rule for the $\vee$ elimination). The only difference to intuitionistic conjunction is the restriction to the same context ($\mathbf{\Delta}$) in the rules for Plus (although you can also have a sequent calculus

presentation of intuitionistic logic where there is only one context).

Recalling the quotation above, an action of type $\mathbf{A} \multimap \mathbf{B} \oplus \mathbf{C}$, when given an action of type $\mathbf{A}$, would produce $\mathbf{B}$ or $\mathbf{C}$, but you do not know which one will be produced before it is actually produced.

- **Of course! (!):** $!\mathbf{A}$ expresses that one has $\mathbf{A}$ without any limitation of resource; that is the situation in mathematics, where the utilization of a lemma is not opposed to its later utilization. In this way, if one saturates the logical formulas using '!' one recovers the principles of classical logic, that then appears as a particular case of linear logic.

  Intuitively, $!\mathbf{A}$ means to have as many copies of $\mathbf{A}$ as necessary. It does *not* mean to have an infinite number of $\mathbf{A}$'s, though. Proof theoretically, $!\mathbf{A}$ means that $\mathbf{A}$ can be produced as many times as needed in the lefthand side of the sequent. The intended meaning of $!\mathbf{A}$ is $\mathbf{A} \otimes \ldots \otimes \mathbf{A}$.

- **Why not (?)**: The meaning of Why not? is difficult because it is linked to the meaning of Par ($\wp$): the intended meaning of $?\mathbf{A}$ is $\mathbf{A} \wp \ldots \wp \mathbf{A}$. $?\mathbf{A}$ means that $\mathbf{A}$ can be produced as many times as needed in the righthand side of the sequent. What does an action $?\mathbf{A} \multimap \mathbf{C}$ mean, for example? Actually, it means that one can use $\mathbf{A}$ or $\mathbf{A} \wp \mathbf{A}$ or $\mathbf{A} \wp \ldots \wp \mathbf{A}$ (and so on) to produce $\mathbf{C}$, but what does this mean? An elucidation of the meaning of $\wp$ would at the same time make the meaning of ? clearer.

2.5.1 Comparisons between Connectives

Besides the above explanations, which are given by people presenting linear logic to beginners, one can also profit from the following comparisons between connectives. These comparisons stress the similarities and differences between connectives according to the features introduced above:

- Multiplicative ($\otimes$) versus additive (&) conjunction: These connectives are the two conjunctions 'created' by the distinction between multiplicative and additive formulations of classical conjunction (*and*). In classical logic, the two formulas joined by the *and* connective may be used as many times as one wishes in a derivation. Linear logic is not so liberal. With the multiplicative conjunction ($\otimes$), each formula must be used exactly once whilst with additive conjunction (&), exactly one of the conjuncts must be used exactly once in a derivation.

- Additive conjuntion (&) versus additive disjunction ($\oplus$): These two connectives allow only one subformula to be used exactly once, but with & you can choose which formula to use (external choice) whilst plus does not allow you to choose (it is already chosen, internal choice, and if you want to prove that something follows from $\mathbf{A} \oplus \mathbf{B}$ then you have to prove that it follows from $\mathbf{A}$ and from $\mathbf{B}$).

- Of course (!) versus Why not (?): These are dual connectives and they have the same characteristic of unlimited reuse but on opposite sides of the se-

quent. They are called *exponentials* because (like $2^a.2^b = 2^{a+b}$) [Sch94]:

$$!\mathbf{A}\otimes!\mathbf{B} \Longleftrightarrow !(\mathbf{A}\&\mathbf{B})$$

$$?\mathbf{A}\wp?\mathbf{B} \Longleftrightarrow ?(\mathbf{A}\oplus\mathbf{B}).$$

Another kind of comparison between connectives is the duality set in the DeMorgan-like equations of Figure 2.14. In Figure 2.5 we present a table showing the dual connectives in linear logic. Dualities are important in order to understand the meaning of connectives mainly because of their role in the cut elimination process. According to Troelstra [Tro??], one can extract computational content from linear logic through the algorithm of cut elimination (as from any other sequent calculus presented logical system). For example, the Cut rule for a composite formula such as $\mathbf{A}\otimes\mathbf{B}$ is

$$\frac{\boldsymbol{\Gamma}\vdash\mathbf{A}\otimes\mathbf{B},\boldsymbol{\Delta}\qquad\boldsymbol{\Gamma}',\mathbf{A}\otimes\mathbf{B}\vdash\boldsymbol{\Delta}'}{\boldsymbol{\Gamma},\boldsymbol{\Gamma}'\vdash\boldsymbol{\Delta},\boldsymbol{\Delta}'}Cut$$

However, in the right-sided formulation of linear logic the above rule would be presented as

$$\frac{\vdash\boldsymbol{\Gamma}^{\perp},\mathbf{A}\otimes\mathbf{B},\boldsymbol{\Delta}\qquad\vdash\boldsymbol{\Gamma}'^{\perp},(\mathbf{A}\otimes\mathbf{B})^{\perp},\boldsymbol{\Delta}'}{\vdash\boldsymbol{\Gamma}^{\perp},\boldsymbol{\Gamma}'^{\perp},\boldsymbol{\Delta},\boldsymbol{\Delta}'}Cut$$

and this rule, thanks to duality, can be rewritten to

$$\frac{\vdash\boldsymbol{\Gamma}^{\perp},\mathbf{A}\otimes\mathbf{B},\boldsymbol{\Delta}\qquad\vdash\boldsymbol{\Gamma}'^{\perp},\mathbf{A}^{\perp}\wp\mathbf{B}^{\perp},\boldsymbol{\Delta}'}{\vdash\Gamma^{\perp},\Gamma'^{\perp},\Delta,\Delta'}Cut$$

$\otimes$	$\wp$
&	$\oplus$
!	?
$\forall$	$\exists$

Table 2.5: Dual connectives in linear logic

Thus, dualities play an important role in the cut elimination process of the right-sided formulation of linear logic and also in *proof nets*, another form of presentation of linear logic proofs [Gir95b], since it is necessary to identify dual formulas in order to specify a cut and remove it.

2.6 Conclusion

The several levels of explanation for the linear logic connectives help to elucidate some aspects of the definition of these connectives. Along with this, the flexibility in the formulation of fragments and the high expressiveness of linear logic are factors that positively influence the acceptance of this logical system by computer science researchers. Also, the features of linear logic, especially resource awareness and the representation of states and transitions, allows it to describe many computer science applications.

However, in our opinion linear logic seems to be inadequate for the representation of computer science applications. This happens because of the three following factors.

First, linear logic is too confuse. That is, it has too many connectives and some of them have an obscure meaning. Second, the high computational complexity of linear logic and its subsystems (fragments) is surely a drawback for its utilization since it makes it difficult to automate reasoning[8]. And third, since linear logic is a closed logical system, it is not easy to perform extensions to it, extensions that might be necessary in order to represent features of applications that are not already represented in linear logic. We will continue to discuss this inadequacy in Chapter 4, where we discuss the works that established a relationship between linear logic and concurrency. In the next Chapter we discuss Concurrency.

[8]First-order logic reasoning, another computationally complex system, is also difficult to automate.

Advantages	**Disadvantages**
More constructive than classical logic	It represents only those aspects of computation related to resource use
High expressiveness	Difficult to understand
Double negation holds without losing constructivity	Negation is only a shift operator
Possibility of formulation of several different fragments	Computationally complex even in the simplest fragments (this can also be seen as an advantage)
Well formulated and innovative proof theory (e.g. proof nets)	

Table 2.6: Linear logic advantages and disadvantages

Chapter 3

Concurrency

3.1 Introduction

Concurrent systems are used in several applications nowadays; for example, in the design of computer network protocols, modelling of database transactions, design of parallel programming languages and distributed systems specification languages. Concurrent systems are systems consisting of processes that can be executed in parallel and communicate with each other. The processing of these systems may be *spatially distributed*; when the processing units are phisically separated. In addition to this, many concurrent systems are also *reactive* systems, i.e. they react to stimuli presented by the environment. There are several kinds of systems in practice which are distributed and/or reactive systems. Therefore, concurrency—the study of the theory and practice of concurrent systems—is a very important subject in computer science.

Mathematical models of concurrency are necessary in order to give precise definitions of the main aspects of concurrent systems. Several models have appeared since 1965 (see Table 3.1): Petri net theory, CCS, CSP and π-calculus, among others. However, many of these present some serious problems; the first problem is the lack of a *typing discipline for concurrency* as good as the existing functional typing disciplines. This typing discipline should, for example, allow the verification of some properties of processes (such as *deadlock freedom*) prior to their execution. The second problem is that there is no well established normal form theorem for processes.

In this work we are mostly concerned with algebraic models of concurrency. These models allow the description of concurrent systems in an algebraic manner. There are many such models in the literature and the representation of features such as set of operators establishes distinctions between them. Here we discuss two good representatives of this kind of model: CCS, the first model in the algebraic approach, and π-calculus, a recent development that is a refinement of CCS and allows the representation of mobile processes.

3.1.1 Outline

In Section 3.2 we discuss mathematical models of concurrency. After that, in Section 3.3 we present the problems associated with these models. Following, we discuss the features of models of concurrency (Section 3.4) and present two algebraic models of concurrency in Section 3.5: CCS (subsection 3.5.1) and π-calculus (subsection 3.5.2). Finally, in Section 3.6 we present the conclusions of

Calculus	Author and Year	Main features
Petri Nets	Carl Adam Petri, 1959	Graphic representation True concurrency
CCS	Robin Milner, 1980	Notion of observation Algebraic approach
CSP	C.A.R. Hoare, 1980	Denotational semantics Programming language approach
π-calculus	Milner, Parrow and Walker, 1989	Mobility Naming

Table 3.1: Most influential concurrency calculi

this Chapter.

3.2 Mathematical Models of Concurrency

Several authors have discussed the necessity of mathematical models of concurrency [Abr84, **?**, Gup94]. In our opinion, mathematical models of concurrency are necessary for three main reasons. First, we need such models because functional and sequential models of computation cannot adequately represent most of concurrent systems. For example, systems that have *time dependency* [Abr84] cannot be represented as functions, because a function always gives the same output for the same input, whilst in a time dependent system the output may vary in time. Such a system could be represented as a function only if time would considered a

parameter, but that is theoretically inadequate. Although it is possible to extend existing functional or sequential models of computation with concurrency features, that would also be inadequate (according to Milner [**?**], we would not have a basic calculus[1]).

Second, we need mathematical models of concurrency in order to be able to formalize the notion of concurrent behaviour. As Gupta rightly points out in [Gup94], "Whenever a programmer writes a program, he or she has some intuitive idea of how the system will behave. For example, if one writes `if (`a `or` b`) then A`, then one has some notion of how the system would evaluate a or b. This may or may not correspond to how the system actually does the evaluation. *The purpose of a mathematical model is to make precise the intuitions, so that there is no gap between the user's perspective and the actual implementation.* This gap is even more apparent in concurrency, making mathematical models absolutely necessary." Therefore, a major contribution of existing models of concurrency is their (tentative) definition of concurrent behaviour in order to reduce this gap.

A third reason for needing mathematical models of concurrency is in order to understand the mathematics behind concurrency. That is, by using mathematical models it is possible to represent formally several features and properties of concurrent systems. For instance, having represented states of concurrent systems and transitions between these states one can formalize the concept of *deadlock state*, which is a state from which there is no transition to any other state. Properties

[1] In CCS, for example, the sequential composition of two processes can be seen as a special kind of parallel composition. Therefore, we cannot have a basic calculus if we already have sequential composition and add another operator for representing parallel composition.

of concurrent systems such as this one can be formalized and, after that, we can reason with descriptions of systems to check if these properties are satisfied or not by the systems.

Every model of concurrency must satisfy some important properties in order to be considered a *good* model. First, any model must allow the description of a great deal of concurrent processes, i.e. must have a great expressive power. This does not mean that it must be able to represent *all* kinds of concurrent processes; this only means that a model must not be *too* restricted to an application area (e.g. the description of computer network protocols) because in this way it is going to reduce significantly its usefulness.

Second, a model must not have a too complicated theory; CCS and π-calculus are good examples — although their semantics is still complicated, these models have a simple definition of their constructors and a great expressive power. Certain models, such as the models whose purpose is to serve as a specification language (e.g. LOTOS), may have a more complicated subjacent theory, given some special requirements such as the conciseness of specifications. However, *basic* models such as CCS, which are used in order to give a better understanding of the main features of concurrent systems, shall be able to represent a great deal of concurrent processes without much complication in the theory.

Third, it must be possible, by using tools provided by a model, to verify properties of the concurrent systems described. In this way, one can verify if the systems satisfy what is expected of them before they are actually built. For example, any

good model must present at least one way to mathematically identify equivalent descriptions of processes (see subsection 3.4.2). And finally, the supporting theory of a model must be adequate, without major problems, in order that one can guarantee that the results obtained in theory hold in practice. For instance, there is no use in proving a congruence between two descriptions of machines if the actual machines constructed from these descriptions cannot be interchanged as component of a bigger system (see subsection 3.4.2).

3.3 Problems of Models of Concurrency

The existence of a great number of models of concurrency nowadays represents a significative improvement in the formal representation of concurrent systems. However, many of these models still present some serious problems, especially when they are compared to sequential and functional models of computation. Here we highlight three problems present in most models of concurrency: *(i)* the lack of an adequate typing discipline for concurrent processes; *(ii)* the absence of normal forms for descriptions of concurrent processes and *(iii)* the indefinition regarding which are the best definitions of equivalence and congruence among processes. Next we discuss these three problems.

3.3.1 Types for Processes

One major problem in the theory of concurrency is to find a good notion of typing for concurrent processes (**typed concurrent programming**). A *typing discipline*

for concurrent processes consists of assignments of *type expressions* to descriptions of processes. These type expressions provide extra information (a kind of specification) not necessarily contained in the description of the process.

In functional computation, typing disciplines offer a great help for the specification of systems. A typed function (ex.: $f : A \rightarrow B$) can receive as parameters only those values or expressions whose type it can handle. In this example, only data values or expressions of type A. If the function receives a value or expression allowed by its type, then it produces a result whose type is also indicated by the type of the function. $f(x)$, of type B, is the result of applying f to a data x of type A.

Typed processes should enjoy some nice properties, like determinacy, convergence and deadlock freedom, according to Abramksy [Abr93]. Deadlock freedom, for example, is related to the possibility of a process entering in a deadlock state. Such a kind of state happens when a process cannot proceed, even though it did not ended its processing. In operational systems theory several possible cases of deadlock are illustrated [Tan92]. In models of concurrency, one can describe several kinds of deadlocked processes, such as, in π-calculus, $(va)(a.0)$ or $(vab)(a.0 \parallel b.0)$. These processes are in deadlock because there can be no reduction from them. Such is not the case for $(va)(\overline{a}.b.0 \parallel a.0)$, than can be reduced to $b.0 \parallel 0$. In a typing discipline for the prevention of deadlock, the two first processes above would not be correctly described, i.e. either no type expression or a type expression representing deadlock would be assigned to them. It is important to notice, however, that in any typing discipline for ensuring some nice property, the set of processes described by the calculus must not be restricted in an excessive way.

3.3.2 Equivalence/Congruence among Processes

In functional and sequential computation models, due to the halting problem [AU69], it is impossible to decide when two descriptions of systems are equal (or equivalent): it is an undecidable problem. However, it is relatively easy to state what makes two system descriptions be the same: in functional models, two systems are equal when they compute the same function, whilst in sequential models two systems are considered equal if they recognize the same language. In models of concurrency, it is also important to be able to identify when two different descriptions of concurrent systems are equivalent in some way. In general terms, two concurrent systems are equivalent if they have the same behaviour. Congruence, which is related to the idea of intersubstitutivity, is a notion stronger than equivalence. For example, suppose that we have a system S where P is a component (this is represented by $S[P]$). If Q is a congruent to P, then $S[Q]$ must be equivalent to $S[P]$, for any system S. That is, exchanging P for Q as component of a system does not affect the behaviour of this system.

For any model of concurrency, several different definitions of equivalence and congruence may be given (see [Mil93]). This happens because of three problems regarding the definition of equivalence and congruence relations. First, it is difficult to define formally what is the *behaviour* of a process. This concept must take into account the sequence of actions performed by a process but not the internal states of the process during execution. A good candidate definition is the definition of observable behaviour presented by Milner in [Mil80]. There, the behaviour of a process is described (informally) as what an (external) observer can see of a process. Second, once you find a formal definition of the behaviour of a process in

a model, then you must test it to see if it serves as a basis to define adequate equivalence relations and also to define normal form theorems. In CCS, for example, observable behaviour leads to the definition of observational equivalence. And third, it is difficult to find some similarity or, even, some relationship between the several different definitions of equivalence and congruence relations in different models or even in the same model. Therefore, one cannot be sure that these are correct. After all, there is nothing like the idea of function in the concurrency world [**?**].

3.3.3 Normal Form for Processes

If any two processes are equivalent (or congruent) but have different descriptions, then it is interesting to be able to *rewrite* these two descriptions to another description equivalent (or congruent) to the former two. This description is called the *normal form* of these descriptions. Although this is an important feature, several models of concurrency do not have a definition of normal form or, if they have, this definition is weak in the sense that not all interesting normal form theorems are proved (see subsection 3.4.7). For example, the reductions that lead to normal form in π-calculus (as defined by Milner in [Mil93]) do not satisfy the Church-Rosser property.

This concept is important in order to facilitate the understanding and finding of equivalences among processes (that in this point of view are seen just as different ways to write the same process). It is also important because it enables a logical study of models of concurrency.

The solution to these problems would make models of concurrency even more useful. Nowadays they are useful because they allow the formal representation of concurrent systems as well as some reasoning with these descriptions. Solving these problems in a model one would get more understanding of the model as well as the possibility of establishing logical foundations via typing discipline and normal forms. In practical terms, it could mean an increase in expressiveness and in the possibilities of formal verification of system properties. However, we are still far from this.

3.4 Features of Models of Concurrency

Why are models of concurrency different from each other? Because, among other things, their *sets of operators* are different, or because some represent *broadcasting* directly whilst others represent indirectly. There are several of these *features* of concurrent systems that can be represented or not in a model of concurrency. And it is the set of features represented in a model that makes models have different expressivenesses, reasoning parts and so on. In summary, these features make possible a distinction between many existing models of concurrent behaviour.

But why certain features are included in some models whilst other features are excluded? First, the set of applications that one intends to represent using a model usually forces the inclusion of some features. For example, if a model is going to represent mobile processes then it will obviously have the mobility feature. Second, although important for some applications, some features may be excluded from a model in order to simplify it, making it easier to understand and to use.

That is, descriptions of processes in that model would have that feature abstracted away. As an example, CCS's descriptions of processes do not take into account the possibility of port names being passed as parameters; only other kinds of data can be passed.

The choices of representation, regarding the features which are included in a model, are made early in the project of that model. And once the design of a model is completed, it is not easily modified; the 'a posteriori' inclusion of features usually leads to badly structured models. Therefore, the appearance of applications requiring new features commonly forces the creation of new models of concurrency. For example, π-calculus has appeared as a refinement of CCS in order to represent mobile processes, which CCS could represent only indirectly.

Therefore, in what follows we are going to discuss some of these features that allow one to differentiate between models of concurrency. Due to the emphasis of our work in algebraic models of concurrency, most of these features are related to this kind of model. Some are only important in the algebraic approach to concurrency (such as set of operators) whilst others apply to all models of concurrency (e.g. equivalences and congruences).

3.4.1 Set of Operators

Any model of concurrency has a set of operators that is used in order to form the processes described by the model. These processes may be constructed from other processes and simpler entities, such as ports, actions, etc. For example, in CCS the parallel composition operator ($\|$) allows one to construct a process $P \| Q$ from the

processes P and Q, whilst the action prefixing operator (.) allows one to construct a process $a.P$ from a port a and a process P.

The set of operators of a model may be *basic* or *elaborate*. In an elaborate set, there are as many operators as necessary for a clearer and more concise specification of systems; it does not matter if there is a degree of redundancy in this set. For example, it is acceptable to have operators $\circ_1$, $\circ_2$, $\circ_3$ such that $P \circ_1 Q \equiv (P \circ_2 Q) \circ_3 \{a\}$, that is, where $\circ_1$ can be derived from $\circ_2$ and $\circ_3$. However, in a basic set of operators one tries to prevent this kind of redundancy. The intention is to have the minimum possible number of operators without sacrificing the intuitive meaning of these operators.

From the basic operators of a model one can define, then, more elaborate operators for specification purposes. But these defined operators will not have the same status in the calculus as the basic operators. For example, the operational semantics will be based only in the basic operators. Therefore, an ideal feature of basic sets, not always achieved, is that no one constructor in the set can be defined from the other remaining constructors.

Independently from being basic or elaborate, the set of operators of a model must have enough expressive power in order to represent a great deal of concurrent processes. That is, although it is almost impossible to have a set of operators that allows a model to faithfully represent any concurrent system existing in practice, it is reasonable to demand that this set represents at least the most well-known applications of concurrent systems, such as, for example, the concurrent systems used in operational systems. Otherwise, it will be a very limited model.

The notion of set of operators is also important because algebraic calculi (such as CCS and CSP) are differentiated from each other by the definition of the set of operators plus the definition of the operational or denotional semantics. For example, if one calculus has a set of operators $(., \|)$ and other has a set $(., +, \|)$, then the second is (probably) more expressive because it has an extra operator: nondeterminism $(+)$. Even if two calculi have the same set of operators (e.g. $(., \|)$), they can be very different if the operational or denotational semantics rules are different.

Although it is possible to design several calculi with different sets of operators, one can see that there is a great similarity between the sets of operators of the most well-known algebraic calculi: CCS, CSP, ACP, π-calculus, etc. This shows that some operators have a very strong intuitive meaning, such as parallel composition and nondeterminism. Of course there are still some differences. For example, in order to represent the sequential composition of processes, ACP has an explicit sequential composition operator $(;)$ whilst CCS represents it indirectly by using action prefixing. Another example of difference between sets of operators is present in [Hen88], where there is the inclusion of another kind of nondeterminism operator: internal nondeterminism $(\oplus)$. CCS nondeterminism $(+)$ is then regarded as an external nondeterminism. Even though there are such differences there are much more similarities than differences between these calculi.

Now we analyse the reasons for the existence of different sets of operators. The definition of a set of operators for a calculus depends on three main factors. First, the concurrency features that are going to be represented by the calculus. Certain features, such as mobility, demand the inclusion of new constructors to the calcu-

lus (see subsection 3.5.2), whilst others can be included by modifying only other parts of the model. Second, the development of an operational or a denotational semantics for the model may indicate the need of new constructors. Such is the case in the simple calculus of [Hen88], where the development of a denotational semantics (acceptance trees) led to the inclusion of a new operation ($\oplus$) to the calculus. In that case, the operator was included in order that every element of the denotational semantics' mathematical structure were assigned to some term of the calculus.

Finally, the intended use of the calculus, either as a specification language or as a basic language, may interfere in the definition of the set of operators. Specification languages such as LOTOS shall obviously have more operators (that is, shall have an elaborate set of constructors) than theoretic languages such as CCS. In the first case it is necessary to have more concise specifications whilst in the second case a smaller set allows a more concise reasoning system.

3.4.2 Equivalences and Congruences

Any model of concurrent behaviour allows the formal representation of concurrent systems existing in the real world. These representations (which we call *processes*) may be considered *equivalent* for certain purposes. For example, two different CCS descriptions of a vending machine may have the same (observable) behaviour for a user. Therefore, this user will regard both machines as equivalent, i.e. as being different implementations of the same machine. It is very important for models of concurrency to define a notion of equivalent processes and also to define how to

decide when two processes are equivalent or not.

In a model of concurrency, the verification of equivalence is not made by analysing the actual machines, but rather, by analysing the descriptions of these machines. As we will see in subsection 3.4.5, a process description abstracts away some details of these machines, representing only the most important aspects. Therefore, two machine descriptions can be considered equivalent ($\equiv$) in several cases, such as:

1. when we have two equal descriptions of equal machines. Ex.: $a.b.0 \equiv a.b.0$.

2. when we have two equal descriptions of two machines that are different only in those details abstracted by the model. Ex.: $a.b.0 \equiv a.b.0$, where the second process performs a and b in a different speed.

3. when we have two different descriptions of the same machine. The difference between the descriptions lies at a syntactic level, such as in $P \parallel Q \equiv Q \parallel P$.

4. when we have two different descriptions of two different machines with the same observable behaviour. The behaviour depends on the abstractions of the model. Ex.: $a.0 \parallel b.0 \equiv a.b.0 + b.a.0$ in CCS (see subsection 3.4.6).

5. when we have two different descriptions of different machines that have the same behaviour only in some specific contexts[2]. Ex.: $x.c.0 \simeq x.c.0 + y.d.0$ in $[\mathcal{P}](\upsilon y)(\overline{x}.0 \parallel \mathcal{P})$ (where $\mathcal{P}$ is a place marker) because $(\upsilon y)(\overline{x}.0 \parallel x.c.0) \equiv (\upsilon y)(\overline{x}.0 \parallel (x.c.0 + y.d.0))$.

[2]A process context is an expression which becomes a process expression if some empty places are filled by a process expression.

Because of these several cases, there are many possible ways to define how two processes can be considered *equivalent* or *congruent*. In spite of this, some basic lines are followed. Two processes are considered equivalent when they have the same behaviour and two processes are regarded as congruent when, besides being equivalent, the susbtitution of one for the other in a bigger processes does not alter the behaviour of the bigger process. That is, P is congruent to Q ($P \cong Q$) if and only if *for all* process contexts $C[]$, $C[P] \equiv C[Q]$.

Then, the differences between the several definitions of equivalence and congruence are based on the definition of behaviour of a process. In CCS [Mil80], Milner considers the behaviour of a process as the sequence of observable actions that a process performs; the internal states of a process and the internal actions performed by the processes do not affect the behaviour, unless they affect the sequence of observable actions. For example, $a.b.0$ is considered equivalent to $a.\tau.b.0$, since (in CCS) τ is the name of a non-observable action.

One of the several ways to identify the behaviour of processes is by using the idea of *simulation*. A process P *simulates* a process Q if and only if for each action that Q does ($Q \xrightarrow{x} Q'$), P can do the same action ($P \xrightarrow{x} P'$) and the remaining process P' also simulates the remaining process Q'. There is a bisimulation between P and Q if and only if P simulates Q and Q simulates P. In this case, we can say that they have the same behaviour. For example, if $P = a.b.c.0$ and $Q = a.b.(c.0+f.0)+e.f.0$, then Q can simulate P (because Q can perform the sequence of actions abc) but not vice-versa (because Q can do e and P cannot, for instance). On the other hand, there is a bisimulation between $a.0 \parallel b.0$ and $a.b.0 + b.a.0$.

The equivalences and congruences between process are very important in models of

concurrency because of the practical implications they have. For example, if two processes are considered congruent, there can be an intersubstitutivity between them. That is, if P is a component of a larger system S and $P \cong Q$, then we can replace P for Q, and the system will remain the same. Other important applications exist and later we shall study CCS and π-calculus equivalences and congruences.

3.4.3 States and Transitions

The state of a system is the snapshot of that system at a particular moment. A transition is an action or event that takes a system from one state to another. These two notions are well-known for computer scientists and practitioners since they play a key role in the description of computer systems. As an illustration, let us see how one can describe a part of the functioning of an operating system by using these concepts.

Suppose that a process, P, wants to use a resource, X. Suppose also that this resource is currently being used by another process, Q, and that the use of this resource is *exclusive* and *non-preemptive*. Therefore, process P must wait until process Q releases X. The initial state of this system is represented in Figure 3.1 as S_1. When process Q releases X (transition t_1) the system goes to state S_2. After that, the system goes to state S_3 when process P acquires control of resource X (transition t_2).

These two concepts are so important that every model of concurrency has some way to express them. Otherwise, an adequate representation of systems would

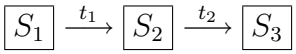

Figure 3.1: Transitions between states

be impossible. There are several different ways to express states in models of concurrency. States can be represented either by the same kind of expression used to describe processes (as in CCS), by a sequence of events (like in some event-based models) or even by a marking of tokens, as in Petri nets.

Transitions, however, have more limited ways of expression. The most important distinction regarding the representation of transitions is the distinction between actions and events. *Actions* are transitions that can occur repeatedly many times during the lifetime of a system whilst *events* are transitions that can never be repeated. In an action-based model, a system can go back to a previous state (where it can repeat a previously performed action), but that is impossible in an event-based model. CCS and Petri nets are examples of action-based models, whilst CSP, Pomsets, Chu Spaces and Event structures [WN95a] are event-based models. Although there is a clear conceptual distinction between these two kinds of model, in practice the difference is smaller since events can be labelled by action names [Gup94].

3.4.4 Operational Semantics versus Denotational Semantics

Besides having a way to describe processes, a model of concurrency must also be able to describe the computational behaviour of processes. This description may be presented in several ways such as, for example, an *operational semantics* or a *denotational semantics* of the model. These two ways of giving meaning to terms of a model have already been successfully applied to models of computation and models of concurrency.

The operational semantics of a model is usually presented as a set of rules, defining a 'reduction' relation ($\longrightarrow$), that represents the computational behaviour of terms. These rules are established according to *intuitions* about the behaviour of the calculus' operators[3]. For example, in CCS's reduction relation the following rule

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}$$

establishes that if there is a process P which in a single computational step (that of doing a) becomes P' ($P \xrightarrow{a} P'$), then the process $P+Q$ can also become P' in a single computational step ($P+Q \xrightarrow{a} P'$), also by doing a. This rule represents the behaviour of CCS processes constructed by using the nondeterminism operator $+$ as the main operator. There are other rules in CCS's reduction relation concerning other operators and special situations, such as communication, are also represented.

A denotational semantics, on the other hand, is a mapping between terms of a model and elements of a mathematical structure. The mathematical structure can

[3]A formal account of operational semantics was given by Plotkin in [**?**].

be other (more basic) model of concurrency or any other adequate mathematical structure. The elements of the structure are said to be the *meaning* or *denotation* of the terms. The assignment from process terms to elements is not freely defined. It must respect, among other things, an equivalence relation between the terms. For example, if $P \equiv Q$, then $[\![P]\!]$ must be the same element as $[\![Q]\!]$. If this happens, we say that the semantics is *fully abstract* with respect to this equivalence relation.

Thus, the establishment of a denotational semantics begins with finding a mathematical structure for this assignment which respects this and other requirements. It is clearly more difficult to establish a denotational semantics for a model than an operational semantics, since the former has to obbey an equivalence relation which is originated in the definition of the latter [Hen88]. For example, Petri nets could not be the denotational semantics of CCS with respect to strong bisimilarity[4], since, for instance, the two strongly bisimilar terms $a.0 \parallel b.0$ and $a.b.0 + b.a.0$ would be naturally assigned to the two *distinct* nets [WN95b] represented in Figure 3.2 and in Figure 3.3, respectively.

3.4.5 Important Abstractions and Ontological Commitments

Models of concurrency are usually designed having in mind some specific set of applications. One may try to be as general as possible in order that the model be also useful for other kinds of systems, but it is almost impossible to achieve a good representation of the *totality* of concurrent systems existing in practice. This happens because when designing a calculus one must make some *abstractions*

[4] A kind of process equivalence.

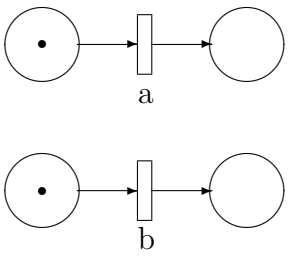

Figure 3.2: Net corresponding to $a.0 \parallel b.0$

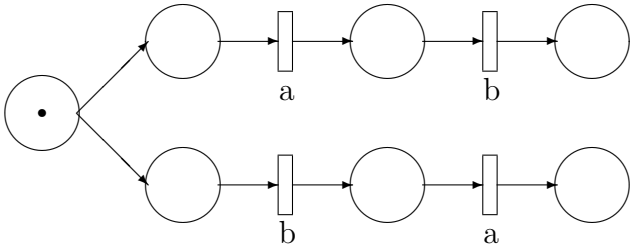

Figure 3.3: Net corresponding to $a.b.0 + b.a.0$

over the reality of systems. These abstractions may be adequate for some systems, but not for others. Therefore, it is not yet possible to have a unified theory of concurrency—different classes of application need to be represented by different models of concurrency.

A calculus would be useless if it did not have some good abstractions for hiding unnecessary details from the programmer or specifier. This happens because it is difficult to reason about the essence of systems having to deal also with these insignificant details. How does one decide which details are insignificant? This is work for the designer of the model. Some details *must* be regarded as insignificant, even though they are not *completely* insignificant, in order to have a more tractable theory, a better formalization of systems and, consequently, a more useful reasoning tool. For example, for most models of functional or sequential programming (such as Turing machines and λ-calculus) the amount of time it takes to execute instructions and programs is considered irrelevant and it is not tackled within the calculus. Therefore, other independent formalisms have been developed in order to reason solely about the time of execution of programs (*complexity measures*). And even those formalisms abstract away from some details of execution time, since only the time of execution of some instructions is taken into account. As concurrency is more complicated than functional and sequential programming, it becomes more difficult to differentiate between the essential and the insignificant details in the processing of a concurrent system. The difference may depend on choices made by the designer of the model.

In spite of this, it is a fact that insignificant details must be abstracted away in the representation. For example, some authors consider that internal actions (actions

between components of a system) should all be considered the same action, since no one observer can see or communicate with these actions. However, these actions can change the internal state and the future behaviour of the machine, therefore they must be represented in a certain way. In CCS, for instance, all internal actions are represented by a silent action τ (greek letter tau). This is a good example of an useful abstraction which is not adopted by all calculi. Another example of abstraction is to consider the actions performed by a distributed system as forming up a sequence of actions (the interleaving assumption—see subsection 3.4.6).

Thus, *ontological commitments* are choices of representation that are made in order to accomplish these and other abstractions. In π-calculus, an example of an ontological commitment is the idea of Naming as a pervasive feature. This idea establishes that one will describe all systems represented by the calculus using only two entities: names and processes. The justification for this is that port names can be passed as parameters to other processes and this allows π-calculus to represent mobility without having to use higher-order, a more complicated feature. Some calculi, however, have ontological commitments that are not made explicit nor justified by their authors.

3.4.6 Interleaving Assumption

In real parallel or distributed systems, two or more actions may occur at the same time. However, some models of concurrency are not able to represent this because of the *interleaving assumption*. In an interleaving model (a model where this assumption is valid) the actions are regarded as *atomic* or indivisble, and even if

two or more actions occur at exactly the same time, the model cannot represent this fact; the model either considers that one occured first and then the other occured or vice-versa. Therefore, in an interleaving model the following law holds: $a.0 \parallel b.0 \equiv a.b.0 + b.a.0$.

CCS, π-calculus and CSP are examples of interleaving models. In CCS, the justification for accepting the interleaving assumption is that there is supposed to be only one observer that can see only one action at a time. Therefore, when observing a process such as $a.0 \parallel b.0$, the observer can see only the sequence of actions ab or ba, even if a and b are performed at the same time. This is added to the assumption that actions are atomic, which eliminates the possibility of actions occuring in overlapping periods of time. In Milner's words: "The reason is that we assume of our external observer that he can make only one observation at a time; this implies that he is blind to the possibility that the system can support two observations simultaneously, so this possibility is irrelevant to the extension of the system in our sense" ([Mil80], page 4).

In a non-interleaving model, however, "events are not projected onto a linear timescale" [Gup94]. Therefore, according to Gupta, "In such a model, $a.0 \parallel b.0$ means that there is no information about the order relation between a and b. This is regarded as different from $a.b.0 + b.a.0$, which represents mutual exclusion between a and b." Chu Spaces [Gup94], as well as Petri nets, Event structures and Pomsets, are non-interleaving models.

Although interleaving models are less expressive than non-interleaving models, they are interesting because the interleaving assumption "leads to a more tractable theory" [Gup94]. This happens because of two factors. First, the only kind of

situation where the interleaving calculus has to handle *directly* the simultaneous occurrence of actions is in communication between processes, where

$$\overline{a}.P \parallel a.Q \xrightarrow{\tau} P \parallel Q$$

Second, with the interleaving assumption one does not need a basic operator for the *sequential composition* of processes, since this operation can be derived from parallel composition, action prefixing and nondeterminism operators. In CCS, for example, if we want to construct a process $P;Q$ where ; is sequential composition, $P = a.0 \parallel b.0$ and $Q = c.0$, then $P;Q$ can be written as $a.b.c.0 + b.a.c.0$ only because of the interleaving assumption.

If we keep the assumption that all actions are atomic and have a special operator only for actions to represent the fact that two or more actions occur at the same time (for example, $a \otimes b$ would describe the action that consists of a and b occurring at the same time), it also becomes possible to represent sequential composition in a model for concurrency; in this way, the sequential composition of P and Q, $P;Q$, would be described as $a.b.c.0 + b.a.c.0 + a \otimes b.c.0$. That is, it would be necessary to have another operator between ports and we still would not be able to represent actions occuring at overlapping periods of time.

Advantages and Disadvantages

The interleaving assumption is a choice, an ontological commitment made by the designer(s) of a model. In this respect, it has some advantages and some disadvantages that must be considered. The main advantages are:

- The calculus becomes simpler and it becomes easier to formalize the reasoning

with concurrent processes, because it usually has less combinators and less operational semantics rules;

- The interleaving assumption may be the *right* level of abstraction for some applications.

However, there are some disadvantages:

- there is a loss of expressive power, since it becomes impossible to distinguish $a.0 \parallel b.0$ from $a.b.0 + b.a.0$;

- the actual interleaving of actions may be unnecessary for most applications. That is, if there is a process described as $a.b.0 \parallel c.d.0$, it may be irrelevant to know that this process performs the sequence of actions *abcd* or *cadb* or any interleaving of the sequences *ab* and *cd*. The only thing that matters is that the subprocesses $a.b.0$ and $c.d.0$ were performed concurrently;

- In real distributed and parallel systems several actions may be performed in a short span of time, some of them can even be performed at the same time. There is no *global clock* which determines the sequence of actions performed by a system. The different speeds of communication channels linking agents may lead each agent to perceive a distinct sequence of actions performed by the system. But in the interleaving assumption it is assumed that the actions occur in a particular order. This can be seen as a drawback of the interleaving assumption, since the fact that actions performed are perceived differently by distinct agents in a system can be significative for reasoning about this system.

- The interleaving assumption does not allow *action refinement*, since it is based on the supposition that actions are atomic (see [Gup94]). This happens because, for example, if in $a.0 \parallel b.0$, a could be refined to $\overline{c}.x$ and b could be refined to $c.y$, then $\overline{c}.x.0 \parallel c.y.0$ would not be equivalent to $\overline{c}.x.c.y.0 + c.y.\overline{c}.x.0$, because, among other things, a possibility of communication, between $\overline{c}$ and c, present in the first term would be lost in the second (interleaved) term;

3.4.7 Normal Form Theorems

In a model of concurrent behaviour, it is usually possible to form different expressions describing the same process. These expressions are considered as describing the same process if they are found equivalent (see subsection 3.4.2). The *normal form* for a process is a process expression in the model to which all equivalent descriptions of a process can be reduced by a *normalisation procedure*, and that is a simpler expression in some *well-defined* sense. Therefore, a normalisation procedure assigns to each process description P a *normal* process description $nf(P)$ such that $P \equiv nf(P)$ and, for all Q such that $Q \equiv P$, $nf(Q) = nf(P)$.

There are several candidate ways for regarding an expression as simpler than another. For example, an expression can be simpler than another because it is the result of some structural transformations (such as, for example, $(a.0 \parallel 0) \longrightarrow a.0$ or $(a.0 \parallel b.0) \longrightarrow (b.0 \parallel a.0)$) operational semantics) on the other process expression. In π-calculus, $x.0 \parallel y.0$ is simpler (in this meaning) than $(va)(\overline{a}.x.0 \parallel a.y.0)$ because the latter expression can be reduced to the former (by performing a silent

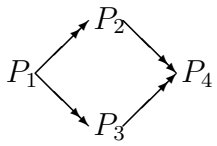

Figure 3.4: Church-Rosser property

action). Another possibility is to consider as simpler an expression that has less symbols. In this case, $a.b.0$ would be considered simpler than $a.b.0 + a.b.0$. Besides that, one could also consider some order of importance between the symbols of the model, and then regard as simpler that expression which has simpler symbols on most important positions. For example, if $+$ is a symbol that is considered simpler than $\|$, then $a.b.0 + b.a.0$ will be regarded as simpler than $a.0 \parallel b.0$.

In order to transform an expression into its normal form, one must develop a number of normalisation rules. In these rules, $P \longrightarrow\!\!\!\!\!\rightarrow P'$ means that P' is closer to $nf(P)$ than P. These rules will have some relationship with the operational semantics and with an equivalence relation of the model. After presenting normalisation rules, one must prove some important theorems. The **weak normalisation theorem** is used to show that for all expressions there is some strategy using normalisation rules that leads to a normal form. The **strong normalisation theorem** shows that *all* normalisation strategies lead to a normal form. And by proving the **Church-Rosser property** one proves the uniqueness of normal form. That is, if $P_1 \longrightarrow\!\!\!\!\!\rightarrow P_2$ and $P_1 \longrightarrow\!\!\!\!\!\rightarrow P_3$, then there exists some P_4 such that $P_2 \longrightarrow\!\!\!\!\!\rightarrow P_4$ and $P_3 \longrightarrow\!\!\!\!\!\rightarrow P_4$, where the P_i's are process descriptions (see Figure 3.4).

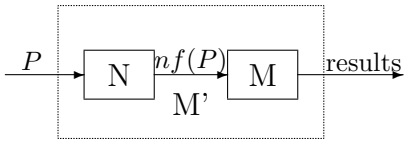

Figure 3.5: Use of normalisation procedure

The existence of a normalisation procedure satisfying the normal form theorems almost guarantees the good computational behaviour of the model. Since normal form descriptions satisfy some properties, it is usually easier to perform computations on them than on general descriptions. Therefore, having a normalisation procedure makes it easier to perform any kind of computation on non-normal process descriptions. For example, if P is a process description and M is a machine that performs some kind of computation with normal forms, then, by using a machine N (that performs the normalisation of process descriptions) one can construct a machine M' that performs the same kind of computation as M, but with all (normal and non-normal) process descriptions (see Figure 3.5).

The normal form theorems can also provide insight about the formalisms on which they are based. In classical logic, for example, the normalisation procedure for sequent calculus (Cut-elimination) shows that one of its rules, Cut, is redundant. That is, every proof with Cut can be transformed into a proof without Cut. In a model of concurrency, a similar result can be obtained. For example, if a certain operator does not appear in the normal form of any process, then it is not basic and it can be removed from the calculus without loss of expressive power.

3.4.8 Action Refinement

Action refinement is a feature of some models of concurrency that has important implications in the description of concurrent systems. In a model that possesses action refinement, it is possible to subdivide an action into several smaller actions in order to further specify details of these actions. For example, in such a model, if a process is described as $P \equiv a.b.0$ and action a can be refined to $c.d.e$, then P will become process $c.d.e.b.0$. In the models where action refinement is absent (CCS, for example), this is not possible because the actions are regarded as atomic.

This concept is important for several reasons. For example, it may be used in order to represent the incremental specification of systems. It enables the specifier to work at several levels of abstraction and to refine specifications making them more concrete. Besides that, if the model is going to be used as a (kind of) programming language (i.e. for programming purposes), then this concept helps the programmer to present his system in several levels of abstraction by enabling the abstraction of irrelevant details. This is important because implementation details can be further modified without changing the interface of a program and also because code may be secret for proprietary reasons.

Although it is a very useful concept, action refinement is not always present in models of concurrency. This happens because its inclusion in a model usually leads to a less tractable theory. For example, in interleaving models it is usually impossible [Gup94] to have action refinement (see subsection 3.4.6). Other problems (such as the necessity of a distinction between refinable and atomic actions) make models that allow action refinement more difficult to design.

3.4.9 Specification versus Implementation Views

A model of concurrent behaviour may be used to present descriptions of concurrent systems in several different levels of abstraction. More abstract descriptions, which basically describe the functionality of a system (i.e. *what* the system does), are commonly called *specifications*. On the other hand, descriptions with plenty of details regarding the functioning of a system (i.e. describing *how* the system works) are called *implementations* [Mil80, Hen88]. There are not precise boundaries separating these two levels of abstraction and, besides that, it is possible to define several other levels in addition to these two. Notwithstanding, the distinction between the specification and implementation levels seems clear and it is well accepted in the literature.

A specification is a very high-level description of a system. In a specification, there are few, if any, details of *how* the system is going to perform; the emphasis is on *what* the system does, i.e. on the observable actions it performs and on its possible courses of action. For example, in order to describe a banking teller-machine system one needs only to describe the interaction between user and machine from the user's point-of-view. There is no need, at this level, to specify what the machine will have to do in order to attend the user's requests.

On the other hand, an implementation is the detailment of *how* the functionality described in the specification is going to be achieved. It consists of a low-level description of a system that can contain, among other things, details regarding the architecture of the system (such as the partition of the system into modules). It is important to notice that when a programmer writes a process description as

an implementation, he may use descriptions of already built systems in order to form new systems, thus reducing the development work. However, this may make it more difficult to understand the process description.

The description of a process as an implementation commonly introduces some internal actions that in some models of concurrency are unobservable (silent). For example, a process specified as $a.b.0$ can be implemented as $(\upsilon x)(a.\overline{x}.0 \parallel x.b.0)$, where the performing of $\overline{x}$ and x cannot be observed since these are internal actions restricted by the restriction operator (υ). It is possible to verify the correcteness of an implementation according to a specification, by using the definitions of equivalence and congruence (see subsection 3.4.2) in the following way: if the two descriptions (an implementation and a specification) are equivalent (or congruent), then we can say that the implementation *satisfies* the specification.

3.4.10 Algebraic View versus Other Views

The features of concurrency that we are discussing in this Section are mostly related to the so-called *algebraic models of concurrency*, since the emphasis in our work is on this kind of model. CCS, CSP and π-calculus are good representatives of the algebraic approach to concurrency. However, there are several others very interesting non-algebraic models, such as Petri nets, Chu spaces and Event structures that have many useful features not present in algebraic models. Here we briefly discuss both classes of model trying to show the advantages and disadvantages of each approach (see Table 3.2).

In an algebraic model of concurrency, processes are represented as terms of an

algebra. These terms are constructed from simple entities and from other terms by using the algebra operators. It is possible to represent the construction of processes from nothing or from other processes. The terms of the algebra are subject to equational laws; this is related to an equivalence relation between processes: equal terms represent processes with the same behaviour. Inequalities between terms can also be used in order to define other relations between processes [Hen88].

A process calculus is defined from a process algebra by defining an operational semantics for the terms of the algebra, thus representing the behaviour of these terms. Derivations and derivatives of terms describe states of processes. Different algebraic models can be defined by varying the set of operators used (see subsection 3.4.1) and/or the operational semantics of the calculus.

The main advantage of algebraic models is that they represent the compositional view of systems. According to Milner, "algebra appears to be a natural tool for expressing how systems are built" ([Mil80], page 4). Besides that, models such as CCS, CSP and π-calculus have a well developed analytic part that allows the verification of several properties of processes. However, there are at least two disadvantages. First, the fact that the visualization of systems is not so good as in Petri nets, for example. That is, it is not so easy to infer from a term what exactly the system does. Second, the development of a theory for finding equivalence between processes is very important and necessary but still has some problems [Mil93] regarding the choice of the right equivalence.

Since models such as Petri nets are not restricted to an algebraic approach, they can present some interesting features that do not exist in algebraic models. In Petri nets, for example, it is possible to represent graphically both systems and

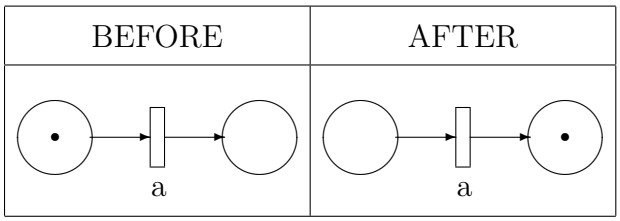

Figure 3.6: Passing of a token

transitions between states of systems. For example, we show in Figure 3.6 how the performing of an action a is depicted, in a Petri net, as the passing of a token (represented as a dot).

A net can also be presented as a mathematical structure (see [WN95a]) and the analytic part of the model is based on this structure (even though some simple verifications can be performed graphically). Petri nets has a great expressive power. It allows a good visualization of systems and it also provides many strong analytic techniques. However, the compositional view of systems is missing; there are not well defined ways to construct a system from other (previously built) systems, i.e. a notion of constructions on nets is missing.

3.4.11 Broadcasting

Broadcasting is a kind of communication in which one process sends a message to be received by a group of processes, instead of only one process. This kind of communication may be used in practice, for example, when one process needs to inform all other processes in a distributed system that a certain peripheral is going to be disconnected, or that a printer is out of paper, among other things. It is also

Model or Class of Models	**Main Advantages**	**Main Disadvantages**
Algebraic Models	Representation of the compositional view of systems	Not so good visualization of systems
Petri Nets	Graphical presentation of systems	Lack of a compositional view of systems

Table 3.2: Advantages and disadvantages of models of concurrency

used in the formalization of communication protocols for computer networks.

Even though broadcasting is important for some applications, it is not represented in some models of concurrency. This happens because models which support this kind of communication are obviously more complicated than models which only support *simple* communication between exactly two processes: a sender and a receiver. Besides that, broadcast communication (between one sender and a *set* of receivers) can be represented as a set of simple communications, each communication in this set being a communication between the sender and one of the receivers in the previous set. Therefore, in some models broadcasting is seen as a derived notion. Of course, it is also possible to represent simple communication as a notion derived from broadcasting. In this case, the set of receivers would consist of only one process.

The representation of this feature in calculi of concurrency makes them more complicated mainly because of two reasons. First, the calculus may have to deal with two or more kinds of communication (since a calculus with only broadcasting would

be limited). This may increase the number of operators and operational semantics rules. Second, newer entities of the model will probably have to be defined in order to support broadcasting. For example, it will be necessary to identify which processes will receive each broadcast message (unless we impose a restriction on broadcast messages, that is, unless we consider only broadcast to all processes—which is not interesting in practice).

In fact, the inclusion of broadcasting in a direct form in basic models is justified only for those models specially designed for the representation of applications where broadcasting plays a key role. For example, languages for the description of concurrent algorithms in environments that have broadcasting as a basic form of communication. In these cases, a notion of broadcasting as derived from simple communication would be inefficient and would generate problems in the analytic part of the model. This feature is also included when its representation does not mean much more complecation added to the final system as, for example, in systems which are already complicated, such as specification languages for distributed systems.

3.4.12 Synchronous versus Asynchronous Communications

A model for concurrency may be purposely designed either for the modelling of systems which interact *synchronously* (synchronous systems) or for the modelling of systems which interact *asynchronously* (asynchronous systems). Synchronous systems are systems that interact in a time-dependent fashion [Mil83], i.e. at every instant of time a transition occurs (see Figure 3.7), whilst asynchronous systems

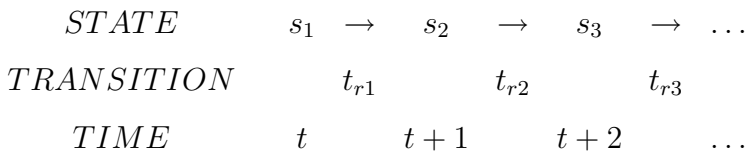

Figure 3.7: Synchronous system

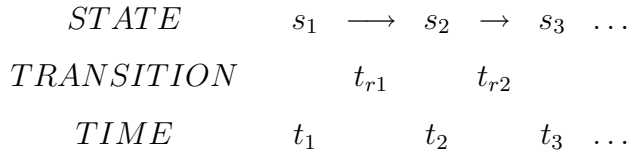

Figure 3.8: Asynchronous system

are systems that may remain in a given state for an unspecified amount of time without performing any transition (see Figure 3.8). For example, in a synchronous model such as SCCS, a process described as $a.b.P$ (where . is action prefixing, a and b are actions and P is a process) will at instant t do a, then in the next instant $(t + 1)$ perform b and then act as P. A similar description in an asynchronous model such as CCS would only amount to saying that the referred process would *eventually* do a and after an *unspecified* amount of time would do b and then become P.

Both kinds of model have specific applications. For example, several kinds of system, such as real-time systems, can only be properly represented as synchronous systems. An advantage of synchronous models is that they can be used to repre-

Idle action	Delay Operator	Representation
1	δ	$\delta E \equiv \text{fix } X(1\text{:X}+E) \quad (X \text{ not free in } E)$
		Derived rules:
		$\delta E \xrightarrow{1} \delta E$
		$\dfrac{E \xrightarrow{a} E'}{\delta E \xrightarrow{a} E'}$

Table 3.3: Definition of delay operator in SCCS

sent both synchronous and asynchronous computations in a single mathematical framework, by defining a *delay* operator and an action to represent idling (an *idle* action). A process that *does* such action is actually idle (see the definition of SCCS's [Mil83] delay operator in Figure 3.3).

However, as pointed out by Milner in [Mil83], the modelling of asynchronous interactions poses simpler problems than the modelling of synchronous interactions. That is so because asynchronous interactions do not have to take into account the time dimension so strictly. And, more important, asynchronous interactions are more adequate to represent distributed programs because "time-dependency is a less prominent feature of programs than of hardware, simply because the essential purpose of a programming language is to insulate the programmer from properties of real computers which may clutter his thinking." [Mil83]

3.4.13 Mobility

A model of concurrent computation that can represent *mobility* allows the description of *mobile processes,* processes that can change their configuration or neighbourhood. A calculus of mobile processes allows the arbitrary linkage of component agents of a system and, in addition to this, this linkage can be changed by the information carried in a communication between neighbour processes. In calculi that do not represent mobility this linkage is fixed, i.e. a process has always the same neighbours throughout its lifetime. The representation of mobility is important for several applications such as, for example, mobile telephone systems (see [Mil93]).

According to [**?**], it is not possible to represent mobility in the most mathematically developed models of concurrency (CCS, CSP, Petri Nets) unless indirectly. That is, the designer of applications which require mobility is responsible for making a correct encoding of such applications in these models. Although there are other models that can express this feature directly, most of them lack a mathematical analysis of their basic concepts. In models such as Hewitt's Actors, mobility is achieved by allowing processes to be passed as values in a communication (processes as data, a higher-order approach). In the higher-order approach, for example, a process Q can send process R to process P. Thus, P becomes able to communicate with R, therefore the configuration of P changes (see Figure 3.9).

Since this higher-order approach makes a model much more complicated, the π-calculus proposed by Milner et alli [**?**] is a model of concurrency that represents mobility without using a higher-order approach. It is based upon a notion of naming [Mil93]: as the communication links are identified by *names*, the π-calculus

$$\overbrace{x(Z).(P' \parallel Z)}^{P} \parallel \overbrace{\overline{x} < R > .Q'}^{Q} \longrightarrow (P' \parallel R) \parallel Q'$$

Figure 3.9: Higher-order approach

$$\overbrace{\overline{y} < x > .P'}^{P} \parallel \overbrace{y(z).Q'}^{Q} \parallel R \longrightarrow P' \parallel Q'\{x/z\} \parallel R$$

Figure 3.10: Mobility in π-calculus

tries to achieve mobility by allowing references to processes (i.e. *links*) to be used in communication. In Figure 3.10 we show a case of *link passing* [**?**]. There, the configurations of three processes change because agent P, which has a link x to R, passes it along its link y to Q; R becomes able to communicate with Q, but no longer with P.

3.4.14 Recursive Definition versus Replication

The so-called action-based models allow the *recursive definition* of concurrent processes. A very simple example is the process $P = a.P$ in CCS, a process that can perform action a indefinetely many times, i.e. the transition $P \xrightarrow{a} P$ can be repeated as many times as necessary. This is a very powerful concept that gives a great expressive power to the models. Besides being the natural way to represent some processes, it also makes the descriptions of some processes more concise,

sometimes making it possible to give a finite representation of processes that in other models would have an infinite representation. On the other hand, it is more difficult to define a denotational semantics (such as Traces for CSP [Hoa85]) for such models (as shown in [Hen88]).

An alternative to recursiveness was presented in Milner's π-calculus [Mil93]: the *replication* operator (!). It is a concept that can substitute recursiveness in many (if not all) situations. In addition to this, it allows a better representation of some situations, such as the copying of machines. The idea is very simple: the process $!P$ means $P \parallel P \parallel \ldots$; as many copies of P as you wish. For example, a process $!a.0$ would have the same behaviour as P of the previous paragraph; it would perform as many a's as necessary because:

$$(!a.0 \equiv a.0 \parallel !a.0) \xrightarrow{a} (0 \parallel !a.0 \equiv !a.0)$$

In [Mil93], Milner shows how to encode other forms of recursiveness by using replication.

Both features can coexist in a model, but basic models such as CCS and π-calculus usually have only one or the other, because having both would be redundant in a certain way and would make the model more complicated. This is another example of abstraction over reality that can have important implications for the resulting models.

3.5 Algebraic Models of Concurrency

There are several ways to approach concurrency (for example, Petri net theory, CCS and CSP). CSP, CCS and π-calculus are representatives of the *algebraic* approach to the description of concurrent systems and here we shall discuss some features and problems of concurrency that are relevant in this approach.

3.5.1 CCS—A Process Algebra

CCS was the first attempt to find basic constructs for concurrency in the algebraic approach. The key idea in CCS is the idea of observation. That is, a system is represented by what an observer can see of it. Although this is in principle an advantage, for it provides the user with the right level of abstraction, this results in a problem because CCS does not represent *true concurrency*. CCS assumes that an observer can only see an action at a time. Therefore it is an *interleaving* model (see subsection 3.4.6). The expressive power of the calculus is thus reduced in a way that can be very harmful for some applications. However, the simplicity of the calculus that is a consequence of this decision of keeping observationality truly compensates, in our opinion, this problem with the representation of concurrent actions.

Some basic notions need to be established in order to understand CCS. A process in CCS is endowed with *ports* through which it may communicate with other processes. A *label* is associated to each port and the modelling of communication between processes is facilitated by the assumption that ports occur in *complementary pairs*. *Input* ports are ports at which a signal or value may be input whilst

output ports are ports at which a signal or value may be output. The labels of output ports ($\overline{a}, \overline{b}, \ldots$) are differentiated from the labels of input ports ($a, b, \ldots$) by marking the labels of the former with an overbar.

One of the most interesting features of CCS is the identification of the basic constructions necessary to represent concurrent systems. The intention is to be able to describe all kinds (or at least several kinds) of concurrent systems using only this set of constructions. In the version of CCS without parameter passing, we have the following constructions to describe agents (see also Table 3.4):

- Zero (0): 0 is an agent. It describes a process which can do nothing. It is used to get constructions started;

- Action Prefixing (.): It is the first constructor of the calculus. Given any label l and agent P we may form the agent $l.P$ which represents a process which may initially perform the action described by l to become P;

- Summation (also called nondeterminism): Given two agents P and Q we may form the composite agent $P + Q$ which represents the process whose capabilities are the conjunction of those of P and Q. Such an agent is called a *sum*, and the operation of forming a sum, *summation*. Thus $+$ is the *summation operator*;

- Composition ($\|$): If P and Q are agents then $P \parallel Q$ is an agent which represents the parallel composition of P and Q. Each one, P and Q, may proceed independently of the other and there is in addition the possibility of communication between them;

Zero process	Action Prefixing	Parallel Composition
$\overline{0}$	$\dfrac{l \quad P}{l.P}$	$\dfrac{P \quad Q}{P \parallel Q}$
Summation	Restriction	
$\dfrac{P \quad Q}{P + Q}$	$\dfrac{P \quad l}{P \setminus l}$	

Table 3.4: CCS formation rules

- Restriction: Given an agent P and a label l, $P \setminus l$, P *restricted on* l, is an agent which behaves like P, except that $P \setminus l$ is unable to communicate through either of the ports labelled l and $\bar{l}$, whereas P may have such a capability.

Several kinds of concurrent systems can be described using these constructors. Some examples of systems in this calculus are:

1. $a.b.0$ is the agent that can perform a and after that becomes $b.0$. In turn, $b.0$ can perform b and after that becomes 0 (which can do nothing);

2. $a.0 \parallel b.0$ is an agent that can do a first and then b or do b first and then a. It does a and b in either order and then stops;

3. $a.0 + b.0$ is an agent that does either a or b and then stops.

The terms of the calculus can also be used to represent the *states* of systems. For example, after performing a the state of the process in item 2 (above) can be represented by the term $0 \parallel b.0$. In addition to this, *transitions* between states are

Action_1	$a.0 \xrightarrow{a} 0$
Summation_1	$\dfrac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}$
Summation_2	$\dfrac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$
Communication	$a.P \parallel \overline{a}.Q \xrightarrow{\tau} P \parallel Q$
Parallel Action_1	$a.P \parallel Q \xrightarrow{a} P \parallel Q$
Parallel Action_2	$P \parallel \overline{a}.Q \xrightarrow{\overline{a}} P \parallel Q$
Restriction	$\dfrac{P \xrightarrow{a} P'}{P \setminus l \xrightarrow{a} P'}$ $(a \neq l$ and $\overline{a} \neq l)$

Table 3.5: CCS reduction rules

represented by the reduction relation of the calculus. The reduction relation is an operational semantics (see subsection 3.4.4) of the calculus in the style of [**?**]. Each rule in the reduction relation represents a computational step. In Table 3.5 we can see the reduction relation of CCS. For instance, the meaning of $\xrightarrow{a}$ is such that if $P \xrightarrow{a} Q$ then the agent P becomes agent Q after performing action a (a computational step).

Using these rules one can reason about properties of processes. For example, in order to verify that the process $(a.0 + x.0) \parallel b.0 \parallel (y.0 + c.0)$ is able to perform the sequence of actions abc, one can use the reduction rules and form the following sequence of reductions (which proves the property):

$$(a.0{+}x.0) \parallel b.0 \parallel (y.0{+}c.0) \xrightarrow{a} 0 \parallel b.0 \parallel (y.0{+}c.0) \xrightarrow{b} 0 \parallel 0 \parallel (y.0{+}c.0) \xrightarrow{c} 0 \parallel 0 \parallel 0$$

3.5.2 The π-calculus

The π-calculus is described by Milner [Mil93] as "a way of describing and analysing systems consisting of agents which interact among each other, and whose configuration is continually changing." The idea is to be able to represent mobile processes not using higher order (processes as data) but instead using the idea of *naming* as the central idea in the theory. That is, instead of passing processes as parameters one passes names of channels (which give access to processes) as parameters.

The π-calculus may be considered as a *refinement* of CCS, since it came from considerations about the excessive number of entities in the latter (especially in the version of CCS with parameter passing). In the π-calculus there are only two entities, names and processes, and everything else (such as data values) is represented as one of these two entities. Therefore, π-calculus is more basic than CCS and has a great expressive power (many examples are available in [Mil93]).

Here we will present the theory of the *monadic* π-calculus (the simplest version of the π-calculus). It is composed of:

- A set of names $\mathcal{N}$. A name is the most primitive entity in the π-calculus. The set $\mathcal{N}$ is assumed to be infinite. Names stand for *channels* or *ports* through which a process may communicate with other processes.
- The other kind of entity is a process, which are $P, Q, \ldots \in \mathcal{P}$ and are built from names by the following syntax:

$$P_i = \Sigma_{i \in I} \pi_i.P_i | P \parallel Q | !P | (\nu x)P$$

I is a finite indexing set; in the case where $I = \emptyset$ we write the sum as $\mathbf{0}$ (a process

which can do nothing). In a summand $\pi.P$ the prefix π represents an *atomic action*, the first action performed by $\pi.P$. There are two basic forms of prefix:

$x(y)$, which binds y in the prefixed process, meaning that
"input some name - call it y - along the link named x"

$\overline{x}y$, which does not bind y, meaning that
"output the name y along the link named x".

Besides *action prefixing* (described above) the other *constructions* of the calculus are:

- *summation* $(+, \Sigma)$*:* $P + Q$ represents the process whose capabilities are the union of those of P and Q;
- *parallel composition* $(\|)$*:* $P \parallel Q$ represents the parallel composition of P and Q where P and Q may communicate with each other or proceed independently of the other one;
- *replication* $(!)$*:* $!P$ means $P \parallel P \parallel \ldots$; that is, one can have as many copies of a process P as one wishes;
- *restriction* (υ)*:* $(\upsilon x)P$ restricts the use of the name x to P. $(\upsilon x)P$ behaves like P, except that it is unable to communicate through either of the ports labelled x and $\overline{x}$, whereas P may have such a capability.

π-calculus reduction relation is not too different from CCS's. It is a relation over processes $(\longrightarrow)$ where $P \longrightarrow P'$ means that P can be transformed into P' by a single computational step. The most important rule is *communication*:

$$(\ldots + x(y).P) \parallel (\ldots + \overline{x}z.Q) \longrightarrow P\{z/y\} \parallel Q$$

Name passing can be illustrated, for example, by the following sequence of reductions:

$$\overline{a}x.\overline{x}z.0 \parallel a(y).y(t).t.0 \xrightarrow{\tau} \overline{x}z.0 \parallel x(t).t.0 \xrightarrow{\tau} 0 \parallel z.0 \xrightarrow{z} 0 \parallel 0$$

where in the first reduction all occurrences of y are substituted by x's and in the second reduction one occurrence of t is substituted by a z. The other rules say that reduction can occur underneath composition and restriction (but not underneath prefix, sum or replication) and that two structurally congruent terms (see definition below) have the same reductions (see other reduction rules in [Mil93]).

3.5.3 Definition of Equivalences in the π-calculus

In the π-calculus, as well as in other formalisms for concurrency, it is a major concern to find good notions of *equivalence* and *congruence* between processes. As an example of the definition of equivalences and congruences in the π-calculus we shall see how Milner defines **reduction congruence** and **strong congruence**. First, he defines two concepts needed in order to define reduction equivalence: unguardedness and observability:

Definition 3.5.1 (Unguardedness and observability) An agent B occurs *unguarded* in A if it has some occurrence which is not under a *prefix* α. A process P is *observable at* α, written $P \downarrow_\alpha$, if some $\alpha.A$ occurs *unguarded* in P with α unrestricted.

Then, Milner defines structural congruence in the following way:

Definition 3.5.2 (Structural congruence) *Structural congruence* ($\equiv$) is a relation over processes in the π-calculus that makes identifications without computational significance. For example, $P + Q \equiv Q + P$ since these processes have different forms but the same behaviour. Structural congruence is needed in order to bring communicands into juxtaposition for the reduction relation.

Now, he is able to define reduction equivalence and congruence:

Definition 3.5.3 (Reduction equivalence) The *reduction equivalence* ($\dot{\sim}_r$) is the largest equivalence relation over processes such that $P \dot{\sim}_r Q$ implies

1. If $P \longrightarrow P'$, then $Q \longrightarrow Q'$ for some Q' such that $P' \dot{\sim}_r Q'$.

2. For each α, if $P \downarrow_\alpha$ then $Q \downarrow_\alpha$.

For example, $x \dot{\sim}_r y$ (abbreviating $x.0$ and $y.0$ by x and y, respectively) but $x \parallel \overline{x} \dot{\not\sim}_r y \parallel \overline{x}$.

Definition 3.5.4 ((Strong) Reduction congruence) It is the largest congruence included in reduction equivalence.

Reduction equivalence and congruence do not have a good definition since quantification over all process contexts is used in the definition of reduction congruence. But, using the notions of structural congruence, respectability and *commitment* relation ($\succ$ – defined for the *polyadic* π-calculus in [Mil93]), Milner gives a very satisfactory definition of congruence. Strong congruence is preserved by every

agent construction of the polyadic version of the π-calculus and two strong congruent processes have the same set of commitments at any time during reduction. In order to achieve this definition, Milner defines simulation and bisimulation in the polyadic π-calculus:

Definition 3.5.5 (Simulation) A relation $\sim_s$ is a simulation if it is respectable and also if $P \sim_s Q$ and $P \succ \alpha.A$, then $Q \succ \alpha.B$ for some B such that $A \sim_s B$.

Definition 3.5.6 (Bisimulation) A relation $\sim_b$ is a bisimulation if both $\sim_b$ and its converse are simulations.

For example, $\overline{x} \parallel y \sim_b \overline{x}.y + y.\overline{x}$ but $\overline{x} \parallel x \not\sim_b \overline{x}.x + x.\overline{x}$. Thus, bisimulation is not a congruence relation since it is not preserved by substitution (of names for names).

Definition 3.5.7 (Strong congruence) P and Q are strongly congruent, $P \sim_{sc} Q$, if every construction of the polyadic version of the calculus preserves bisimulation.

For example, if $X \sim_{sc} Y$ and $y.[]$ is a construction, then it is true that $y.X \sim_{sc} y.Y$. Milner has shown that this holds for all polyadic π-calculus constructions [Mil93].

By looking at the definitions above one is able to see how it is possible to define several different equivalence and congruence relations in the polyadic π-calculus. Since most of these relations are meaningful, it may become difficult to decide which one is best for practical or theoretical purposes. In our opinion, a logical approach to concurrency may provide an answer to the question of finding the right equivalence and congruence relations.

3.6 Conclusion

In this Chapter we have studied concurrency — the study of the theory and practice of concurrent systems. Concurrency is a very important subject in computer science and has many applications — for example, it is used in the representation of spatially distributed and reactive systems. In order to describe concurrent systems we need *mathematical* models of concurrency. Besides representing systems, these models allow one to reason about properties of concurrent systems. Here, we have centered our attention in algebraic models of concurrency and have discussed some of the features that are important for the definition of such models.

The models of concurrent computation are relatively new compared to sequential and functional models of computation. Petri nets, one of the oldest models of concurrency, has appeared in 1965, whilst Turing machines and λ-calculus (models of functional computation) were developed in the 1930's. CCS and π-calculus are even more recent (1980's-1990's). Therefore, although these models have some good features, it is natural to expect some problems, especially when they are compared to sequential and functional models. Two problems are the most important in our opinion: most models lack a good typing discipline as well as normal form theorems. Besides that, for some models there has not been consensus regarding which are the right notions of equivalence and congruence relations. However, as we have said, this is acceptable since the models were not developed such a long time ago.

After discussing models of concurrency in general, their problems and features, we have presented two of these models: CCS and π-calculus. Both are algebraic

models that allow the representation of a great deal of concurrent systems, as well as formal reasoning with descriptions of systems and the definition of equivalence relations. In the next Chapter we will discuss the works relating linear logic and concurrency that use these models in order to present possible solutions to some problems of concurrency.

æ

Chapter 4

Linear Logic and Concurrency

4.1 Introduction

In this chapter we will discuss some of the several approaches used for obtaining a logical foundation to concurrency through a relationship between logical systems and models of concurrency. The emphasis here is on the 'proofs as processes' paradigm, which relates linear logic to algebraic models of concurrency.

First, we will discuss briefly the 'propositions as types' paradigm, an important result in the interaction between logic and computation that establishes a relationship between intuitionistic logic and functional and sequential computational models. After that, we will show why classical logic has not been widely used for representing concurrency. Next we discuss the interaction between modal logic and concurrency as well as the results of a recent work in this approach.

Finally, we study Abramsky's adaptation of the 'propositions as types' paradigm

to the concurrency world: the 'proofs as processes' paradigm, and we analyse Bellin and Scott's work [BS94], which contains the most important results of this paradigm.

4.2 Intuitionistic Logic and Functional and Sequential Computation

There are several ways to perform the interaction between logic and computation [Gab90]. For functional computation in particular, one of the most developed approaches is the 'propositions as types' paradigm. In this paradigm the most interesting result is the 'Curry-Howard isomorphism' (CHI) (see Chapter 5, Section 5.3) [How80, dG92, Gir88, Abr94a], which establishes a close correspondence between intuitionistic logic and a functional calculus in the style of the λ-calculus (a well-known model of functional computation).

In the Curry-Howard isomorphism, to each intuitionistic proof is assigned a functional term (called the *functional interpretation* of the proof) in such a way that the notions of *conversion, normality* and *reduction*, introduced independently in the two cases, correspond perfectly [Gir88]. That is, if P is a proof to which is assigned a functional term $fi(P)$, and if $P \longrightarrow P'$ (in the logic's reduction relation) then it must be possible to have $fi(P) \longrightarrow Q$ (in the functional calculus reduction relation) in such a way that Q is equivalent to the functional interpretation of P' ($fi(P')$) (see Figure 4.1). Therefore, if a term is reduced to normal form, the accompanying proof may also be reduced to its normal form, and vice-versa (see

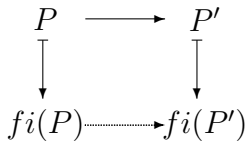

Figure 4.1: The Curry-Howard isomorphism

$$\mathrm{APP}(\lambda x.f(x), a) \longrightarrow f(a)$$

$$\dfrac{\dfrac{[\mathbf{A}]}{\vdots}\ \mathbf{B}}{\dfrac{\mathbf{A} \Rightarrow \mathbf{B} \qquad \mathbf{A}}{\mathbf{B}}} \longrightarrow \begin{array}{c}\mathbf{A}\\ \vdots\\ \mathbf{B}\end{array}$$

Figure 4.2: A reduction in the functional calculus and the corresponding reduction in the logical calculus

Figures 4.2 and 4.3). This isomorphism is relevant because, among other things, it provides a logical foundation for functional programming typing disciplines[1].

[1] "Type checking in functional programming is employed to constrain the use of functional application, guaranteeing 'compatibility' of function and arguments, and hence good behaviour of well-typed programs (e.g., strong normalisation). Type checking is probably one of the most successful application of 'formal methods' to date" [Abr94a]. For example, if $f(x)$ is of type $\mathbf{A} \Rightarrow \mathbf{B}$ and a datum a is of type $\mathbf{A}$, then the result of applying the former to the latter is $\mathrm{APP}(f(x), a)$, which is the same as $f(a)$, of type $\mathbf{B}$.

$$\dfrac{\dfrac{\begin{array}{c}x : [\mathbf{A}] \\ \vdots \\ f(x) : \mathbf{B}\end{array}}{\lambda x.f(x) : \mathbf{A} \Rightarrow \mathbf{B}} \quad a : \mathbf{A}}{\mathrm{APP}(\lambda x.f(x), a) : \mathbf{B}} \quad \longrightarrow \quad \begin{array}{c} a : \mathbf{A} \\ \vdots \\ f(a) : \mathbf{B}\end{array}$$

Figure 4.3: Reduction with functional terms in the labels accompanying proofs

Other approaches have also provided some important results in the interaction between logic and sequential and functional computation. For example, Labelled Natural Deduction [dG92] has generalized the functional interpretation to other logics beyond the realm of intuitionism [dG92, Gd92] by extending the 'propositions as types' paradigm to handle other problems not tackled by CHI. Another example is the works of Abramsky [Abr93], Lincoln [Lin92b] and others, which have provided functional interpretation for intuitionistic subsystems of linear logic, resulting in the definition of *linear* (resource conscious) functional calculi.

4.3 Classical Logic and Concurrency

In an intuitionistic system there is at most one formula in the right side of a sequent and possibly several formulas in the left side. That is, an intuitionistic sequent has usually the following form: $\mathbf{A_1}, \ldots, \mathbf{A_n} \vdash \mathbf{B}$, possibly not existing a $\mathbf{B}$. This asymetry between assumptions (inputs) and conclusions (outputs) is well

suited to represent functional programming, because Cut in intuitionistic logic is non-commutative and the application of functions to arguments is "probably the most important example of a non-commutative operation" [Ale94].

Concurrency, however, is not restricted to one output. A concurrent process can interact with several other processes. Therefore, intuitionistic logic seems to be inadequate to represent concurrency. Since classical logic sequents are not restricted to have at most one formula at the right side, it seems more natural to think of classsical logic in order to represent concurrency.

However, classical logic reduction relation is not confluent; that is, the cut elimination algorithm for classical logic does not have Church-Rosser property[2]. For a given proof, several 'different' cut-free proofs can be obtained by the algorithm. Thus, all of these cut-free proofs of a given sequent $\mathbf{\Gamma} \vdash \mathbf{\Delta}$ have to be identified in order that the cut-elimination algorithm remains confluent, but this is inconsistent with an algorithmic view of proofs (see [Gir88], Ap. B).

The facts above amount to saying that in order to use classical logic in a relationship with a computational formalism one must make (serious) restrictions to the latter; that is, "classical logic is not constructive enough" in order to represent concurrency [Ale94]. Besides that, one can also notice that classical logic formulas are not able to express concisely and adequately all concurrency computational features (see Chapter 3) because its connectives do not have a fine computational meaning. Hence, modal and linear logic have been more frequently used instead of classical logic in order to represent concurrency.

[2] "The result of reducing a proof (...) depends on the order in which reductions are performed, so nondeterminism is introduced." [Ale94]

4.4 Modal Logic and Concurrency

The idea of using modal logic for the representation of concurrency arises almost naturally when one studies process algebras such as CCS, CSP and π-calculus. For example, a process such as $a.b.0 + a.c.0$ will *necessarily* perform action a, whilst another process $a.0 + c.0$ will *possibly* perform action c. From observations such as this one it is possible to define modalities specifically to represent concurrency; instead of the modality of necessity, namely '$\Box$', we would have $\boxed{\alpha}$ (representing that action α will necessarily be performed) and instead of $\Diamond$ (possibility) we would have $\diamondsuit_{\alpha}$ (representing that action α will possibly be performed). Besides that, derived notions such as *may* and *must* [Hen88] can be defined and modal formalisms may be created in order to specify concurrent processes.

Amongst the works in the approach relating modal logic to concurrency some of the most important are: Hennessy and Milner's "Algebraic Laws for nondeterminism and concurrency" [HM85], Hennessy's "Algebraic Theory of Processes" [Hen88] and Amadio and Dam's "Toward a modal theory of types for the π-calculus" [AD96], which we discuss in subsection 4.4.1. The approach defined in [HM85] is used in several other works, such as [Mil93] and [AD96]. A somehow different approach is presented in [Hen88], where the *may* and *must* relations are used in order to prove some equivalences and congruences between processes, providing also a denotational semantics for a process algebra (similar to CCS) and a proof system which is sound and complete regarding this semantics and the notions of congruence defined.

4.4.1 A Recent Work on Modal Logic and Concurrency

A recent work in the approach that uses modal logic in order to type concurrent processes is "Toward a modal theory of types for the π-calculus" [AD96]. The idea is to use an extended version of the modal μ-calculus (a 'Hennessy-Milner logic') in order to type processes described using a version of the monadic π-calculus without replication but with recursive definition.

The main idea is to have an *interpretation* that assigns to each μ-calculus formula (ϕ) a set of π-calculus processes ($[\phi]$), such that if $p \in [\phi]$, then $\models p : \phi$. After that, a model checker is defined in order to verify if a process satisfies a given specification, that is, if a process p is contained in the set that is the interpretation of a formula ϕ. If this can be verified using the model checker, then we say that $\vdash p : \phi$. Next, a *logical equivalence* between processes ($\sim_{\mathcal{L}}$) is defined in the following way:

$$p \sim_{\mathcal{L}} q \text{ if and only if } \forall\phi(\models p : \phi \text{ if and only if } \models q : \phi)$$

where $\models p : \phi$ if p is in the interpretation of ϕ. Finally, this work also presents a *proof system* supporting the compositional proof of process properties. In this system, from the properties of component processes ($x_1 : \phi_1, \ldots, x_n : \phi_n$) one can prove the property of a bigger process ($p : \phi$, where the x_i are in the expression p). When this happens, we say that $x_1 : \phi_1, \ldots, x_n : \phi_n \vdash p : \phi$.

The following results were obtained: for the given interpretation the model checker establishes an algorithm which is sound in general (i.e. if $\vdash p : \phi$ then $\models p : \phi$) but complete only for a restricted class of processes: the processes which have the *finite reachability* (FR) property (i.e. if $\models p : \phi$ and p is FR, then $\vdash p : \phi$); nothing

is guaranteed if p is not FR and $\vdash p : \phi$. Regarding the logical equivalence, it is demonstrated that $p \sim q$ (where $\sim$ is bisimulation in the used version of the π-calculus) if and only if $p \sim_{\mathcal{L}} q$. Finally, the compositional proof system presented in the paper is demonstrated being capable of proving only a restricted set of process properties: non-recursive properties (i.e. properties not involving the recursion operator of the μ-calculus).

Besides these restrictions in the results, we can highlight some other problems in this work. First, the μ-calculus is not a logic but a logical formalism; therefore it has less intuitive meaning than a logic and the notion of *proof* does not have the same importance as in intuitionistic and classical logic. Second, the results obtained are weaker than CHI; for example, the types are not used to constrain the composition of processes. However, as the title of the paper indicates, this is just an initial work and even these restricted results are very interesting and promising at such a stage.

This approach of using modal logic in order to understand concurrency arises quite naturally from observations of process algebras and has produced some positive results, such as the definition of an equivalence relation based on logical considerations which is equal to the bisimulation defined for the π-calculus [Mil93, AD96]. However, as we have already said, the results obtained are weaker than CHI since the formalisms lack two related notions: an algorithmic side and a good notion of proof. Also, a higher intuitive meaning (from a logical viewpoint) for these formalisms is missing; they mirror the process algebras they represent. Finally, the practical results obtained (in terms of implemented proof systems) are still weak,

but there are some ongoing works trying to improve these results (see [AD96]).

4.5 Linear Logic and Concurrency

The most developed way to establish a relationship between linear logic and concurrency is through the 'proofs as processes' paradigm developed by Abramsky [Abr93, Abr94a] and used by Bellin and Scott in [BS94]. Following the suggestion made by Girard that linear logic derivations in some formalism (sequent calculus or proof nets[3]). The process term, which may be viewed as the *computational interpretation* of the proof, is constructed in such a way that there is a close relationship between the meaning of the connectives used in the derivation and the constructions used in the formation of the process. Besides that, there must be a close correspondence between the normalisation of proofs and the reduction of terms in order to establish an isomorphism in the style of Curry-Howard isomorphism.

Abramsky presented two insights aiming at achieving such an isomorphism. The main insight consists of seeing parallel composition as the natural computational interpretation of Cut in linear logic (since in its right-sided sequent formulation the Cut rule may be considered to be commutative and associative). That is, if $\vdash \mathbf{\Gamma}, \mathbf{\Delta}$ is the same (because of the existence of Exchange rule) as $\vdash \mathbf{\Delta}, \mathbf{\Gamma}$, then $\dfrac{\vdash \mathbf{\Gamma}, \mathbf{A}^{\perp} \qquad \vdash \mathbf{\Delta}, \mathbf{A}}{\vdash \mathbf{\Gamma}, \mathbf{\Delta}}$Cut and $\dfrac{\vdash \mathbf{\Delta}, \mathbf{A} \qquad \vdash \mathbf{\Gamma}, \mathbf{A}^{\perp}}{\vdash \mathbf{\Delta}, \mathbf{\Gamma}}$Cut are also the same thing. Therefore, if one assigns a process term P to the proof $\vdash \mathbf{\Gamma}, \mathbf{A}^{\perp}$, a process term Q to the proof $\vdash \mathbf{\Delta}, \mathbf{A}$, and an operation $(\circ)$ is the computational interpretation of the Cut rule, then the result of both proofs ($P \circ Q$ and $Q \circ P$, respectively) must

[3] A new formalism for linear logic proofs developed by Girard in [Gir87].

be considered equal. That is true when $\circ$ is parallel composition.

The second of Abramsky's insights is to view one-sided sequents as *interface specifications* for concurrent processes. The term assigned to each proof of a sequent records the set of cuts that has been performed during the proof:

$$\vdots$$
$$P_{x_1,\ldots,x_n} : \ \vdash x_1 : \mathbf{A_1}, \ldots, x_n : \mathbf{A_n}$$

The free variables (port names $x_1, \ldots, x_n$) in the process term $P_{x_1,\ldots,x_n}$ are exactly the variables annotated to the formulas in the sequent. The $\mathbf{A_i}$'s "constrain how the interface ports labelled $x_1, \ldots, x_n$ can be plugged into corresponding ports in the environment of the process" [Abr94a]. Thus, Cut is viewed as "parallel composition + hiding/restriction": $(P \| Q) \setminus x$ in CCS or $(P \parallel Q)/x$ in CSP. The practical implications of these insights are discussed in [Abr94a].

In [BS94], Bellin and Scott presented an adaptation of Abramsky's translation mapping proofs in linear logic into terms of the π-calculus[4]. They have used the *synchronous* π-calculus: a version of the π-calculus developed by Robin Milner purposely supporting some of the logical rewritings envisioned in [Abr93]. Bellin and Scott obtained some fine results using the 'proofs as processes' paradigm when representing a fragment of the π-calculus by the multiplicative fragment of linear logic (MLL). However, for more complicated fragments (such as MALL and complete propositional linear logic) the results were not as satisfying as CHI.

Several problems have appeared when it was attempted to establish an isomorphism between cut elimination algorithm and process calculus reduction relation.

[4]The translation of linear logic proofs is made according to rules given in [BS94].

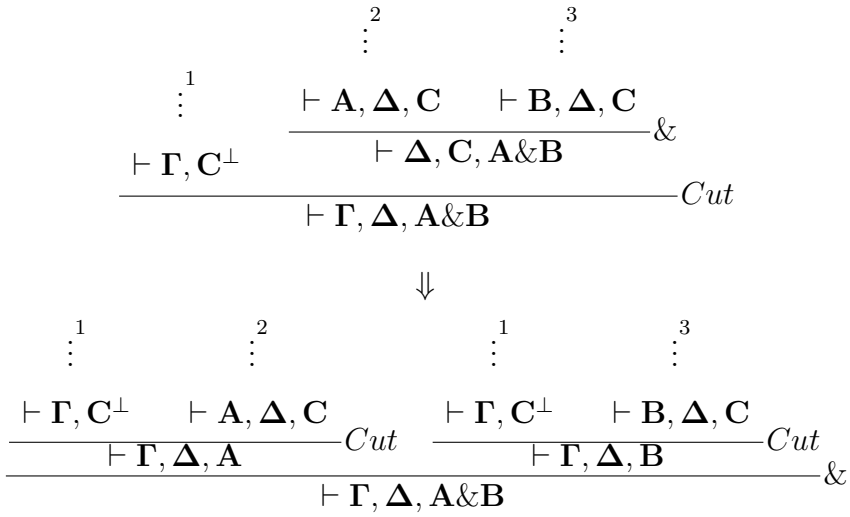

Figure 4.4: Additive commutative reduction in linear logic

The more problematic reduction of linear logic proofs in the cut elimination process, from the point-of-view of the relationship between linear logic and concurrency, is the *additive commutative reduction* (see Figure 4.4). This reduction does not eliminate any cut, but rather pushes one cut up, transforming it into two cuts. These two cuts will be further eliminated in the cut elimination algorithm by other reductions.

The additive commutative reduction annotated with variables accompanying the formulas and process terms alongside sequents, according to Bellin and Scott's modification of Abramsky's translation, is presented in Figure 4.5.

If the *synchronous* π-calculus reduction relation would reflect the additive commutative reduction, then the two terms alongside the conclusion sequent of each proof would have to be either reducible to each other or congruent. But the first

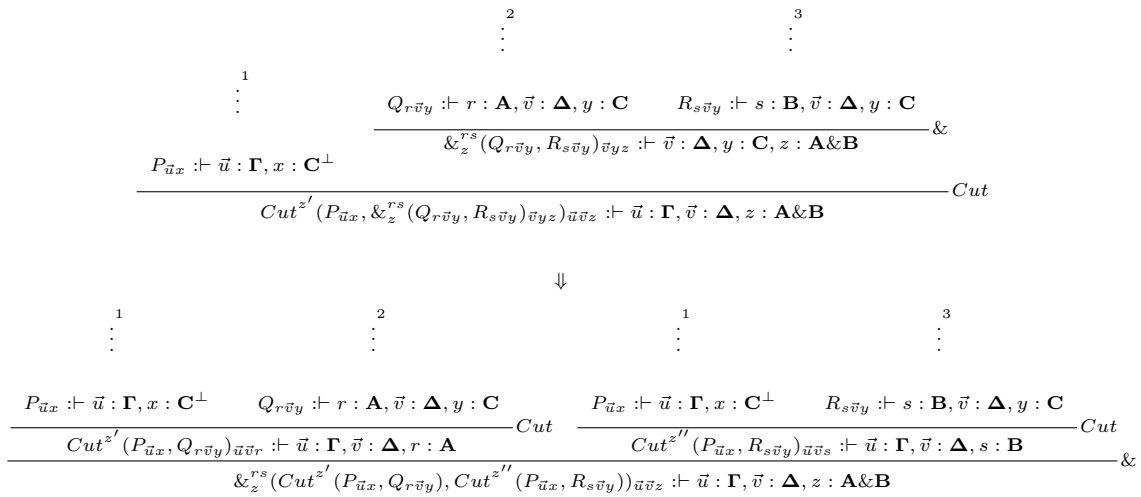

Figure 4.5: Additive commutative reduction with terms

term reduces in this way (using structural congruence several times):

$$Cut^{z'}(P_{\vec{u}x}, \&_z^{rs}(Q_{r\vec{v}y}, R_{s\vec{v}y})_{\vec{v}yz})_{\vec{u}\vec{v}z} \equiv$$

$$\upsilon z'(P_{\vec{u}x}[z'/x] \parallel \&_z^{rs}(Q_{r\vec{v}y}, R_{s\vec{v}y})_{\vec{v}yz}[z'/y])_{\vec{u}\vec{v}z} \equiv$$

$$\upsilon z'(P_{\vec{u}z'} \parallel (\upsilon ab)\overline{z}\langle ab\rangle(a(r)Q_{r\vec{v}z'} + b(s)R_{s\vec{v}z'}))_{\vec{u}\vec{v}z} \equiv$$

$$\upsilon z'(\upsilon ab)(P_{\vec{u}z'} \parallel \overline{z}\langle ab\rangle(a(r)Q_{r\vec{v}z'} + b(s)R_{s\vec{v}z'}))_{\vec{u}\vec{v}z} \equiv$$

$$\upsilon z'(\upsilon ab)\overline{z}\langle ab\rangle(P_{\vec{u}z'} \parallel (a(r)Q_{r\vec{v}z'} + b(s)R_{s\vec{v}z'}))_{\vec{u}\vec{v}z}$$

whilst the second term reduces in the following way:

$$\&_z^{rs}(Cut^{z'}(P_{\vec{u}x}, Q_{r\vec{v}y}), Cut^{z''}(P_{\vec{u}x}, R_{s\vec{v}y}))_{\vec{u}\vec{v}z} \equiv$$

$$\&_z^{rs}(\upsilon z'(P_{\vec{u}x}[z'/x] \parallel Q_{r\vec{v}y}[z'/y]), \upsilon z''(P_{\vec{u}x}[z''/x] \parallel R_{s\vec{v}y}[z''/y]))_{\vec{u}\vec{v}z} \equiv$$

$$(\upsilon ab)\overline{z}\langle ab\rangle[a(r)\upsilon z'(P_{\vec{u}z'} \parallel Q_{r\vec{v}z'}) + b(s)\upsilon z''(P_{\vec{u}z''} \parallel R_{s\vec{v}z''})]_{\vec{u}\vec{v}z} \equiv$$

$$(\upsilon ab)\upsilon z'\upsilon z''\overline{z}\langle ab\rangle[a(r)(P_{\vec{u}z'} \parallel Q_{r\vec{v}z'}) + b(s)(P_{\vec{u}z''} \parallel R_{s\vec{v}z''})]_{\vec{u}\vec{v}z} \equiv$$

$$(\upsilon ab)\upsilon z'\upsilon z''\overline{z}\langle ab\rangle[(P_{\vec{u}z'} \parallel a(r)Q_{r\vec{v}z'}) + (P_{\vec{u}z''} \parallel b(s)R_{s\vec{v}z''})]_{\vec{u}\vec{v}z}$$

Therefore, the translation would be correct if and only if the last terms in the reduction sequences were equivalent. But this is not the case here because this would only happen if there was distributivity of $\parallel$ over $+$, i.e. if $P \parallel (Q + R) \equiv (P \parallel Q) + (P \parallel R)$ (and this is not true because the two processes are not congruent; when composed with a third process the resulting compositions have different behaviours).

Bellin and Scott's own translation make only some minor modifications to Abramsky's translation trying to solve the problems with additive commutative reduction. First, they introduce a new version of the π-calculus, the *full synchronous* π-calculus. The *full synchronous* π-calculus is an adaptation of the *synchronous* version that permits both guarded and unguarded prefixes, where unguarded prefixes can be permuted up to $\equiv$ in certain cases, whilst guarded prefixes can never be permuted nor can reductions occur underneath them. Unguarded prefixes are used in the translation of the multiplicatives, and guarded prefixes are used in the translation of the additives. The problem in Abramsky's translation is solved in the following way: in Bellin and Scott's own translation the additive commutative reductions cannot be represented in the process calculus — they are prevented by guarded prefixes.

Besides this improvement on Abramsky's work, Bellin and Scott have also shown that this kind of translation is essentially about the abstract pluggings in proof structures. In their own words: "communication in the π-calculus, insofar as it

relates to linear logic, is really only about the pluggings in proof structures, not the logic itself". Since proof structures are more general than proof nets[5], the presentation of the results is more complicated. This is compensated by the fact that the results are more general and also because in this more general setting Bellin and Scott were able to formulate many other results [BS94].

4.5.1 Reflections on the 'proofs as processes' Paradigm

Linear logic has been used in the recent works relating logic and concurrency because it is considered *more computational* than classical logic. This has happened inspired by (i) the failure of some tentatives of giving a good account of the logical content of concurrency in terms of classical logic; (ii) by the works relating linear logic and functional programming on the style of the Curry-Howard isomorphism, which showed that linear logic connectives and proof theory indeed have a finer computational meaning; (iii) by the good features enjoyed by linear logic proof theory (unlike other logics that appeared in the recent proliferation of logics motivated by the interaction with computation); (iv) by the sound and faithful translation of several machines into linear logic in the proofs of complexity measures of its fragments; and mostly, (v) by the resource use awareness of linear logic.

But there are still some open questions concerning the use of linear logic for understanding concurrency. First, to treat concurrency (or even computation in general) logically is it enough to have resource use awareness in the logical system? Sec-

[5]A proof net is a proof structures that is sequentializable; it corresponds to a proof in linear logic sequent calculus.

ond, wouldn't it be better to handle features such as resource use in the meta-level, therefore using a deductive system that has a special concern for the handling of meta-level features, such as *LDS* (see Chapter 5)? (a good reason for this would be to simplify the connectives, that in linear logic are quite difficult to understand; another reason would be to allow at the same time the handling of other features, which may be relevant to other computation applications, in the meta-level).

Linear logic is more *computational* than classical logic because its formulas (and connectives) carry information about the resource use of its subformulas. This kind of information is not considered in classical or intuitionistic logic. However, resource counting is only one of many features of computational behaviour. Therefore, linear logic is somehow *limited* to this view of computation. Since linear connectives carry information about the resource use of formulas in a derivation, it seemed a good idea to use them to model basic concurrency constructs (which are not well modeled by classical connectives and to which resource use is a key feature). But because of the mixture between logical features and meta-level features these linear connectives do not seem to have a strong intuitive meaning and this limits their use.

The connectives of linear logic are at a (computationally speaking) lower level than their classical or intuitionistic counterparts, since its is possible to give a sound and faithful translation of the latter in terms of the former. The best example is intuitionistic implication ($\Rightarrow$). $\mathbf{A}\Rightarrow\mathbf{B}$ can be translated into $!\mathbf{A}\multimap\mathbf{B}$, meaning that $\mathbf{A}$ can be used as many times as one needs in order to obtain $\mathbf{B}$. The converse is not possible, that is, it is not possible to express in a simple and faithful way the linear implication in terms of intuitionistic implication. The problem is that linear

implication $\mathbf{A} \multimap \mathbf{B}$ means that $\mathbf{A}$ must be used *exactly once* in order to obtain $\mathbf{B}$. But what does this mean in face of the fact that $!\mathbf{C} \multimap \mathbf{B}$ allows one to use $\mathbf{C}$ as many times as one wish? What does it mean to use $!\mathbf{C}$ exactly once? Actually it means to use $\mathbf{C}$ many times, explictly copying and discarding when necessary. This is a bit confusing.

In the works that analyse the computational complexity of the decision problems for linear logic fragments, several machines have been given a translation into these fragments showing the expressive power of linear connectives and rules. This is another proof of the computational content of linear logic, but since some of these *logical* translations are rather unnatural and do not give any new insight about the machine, it remains to be seen whether the connectives are really basic computationally speaking.

Linear logic proof theory has cut-elimination theorems for all fragments, unless those that arise from very strange modifications. In addition to this, the system is constructive even though double negation holds. Other interesting features are valid for the system (see [Gir87, Gir95b]), but linear logic has appeared (or, rather, it is explained as if it had appeared) as a technical modification of the sequent calculus for classical logic, not as the result of any meta-level consideration about logic itself. And this can be seen as a weakness of linear logic.

Linear logic does not have natural deduction (because natural deduction does not allow multiple conclusions) but has proof nets as a graphical tool to present its proofs. Proof nets enjoy some features as, for example, that the order of application of rules does not matter in certain cases, although in sequent calculus formulation it matters—that is, two sequent calculus derivations may be associated to the

same proof net. This allows the identification of several proofs that are equal up to the order of application of rules. But as a logical formalism, proof nets are very difficult to understand and it is not very easy to check whether or not a given proof is correct. The resource use features of linear logic are the main responsibles for these difficulties.

One of the Abramsky's objectives was to show that the process calculus that was going to be exhibited as the computational counterpart of the proof system was sufficiently expressive in order to allow a reasonable range of concurrent programming examples to be handled, in analogy to the situation with typed functional languages. Bellin and Scott's translation of linear logic proofs into π-calculus, however, exhibits meanings of proofs as some very special process terms. Besides that, some very important reductions (like some of *multiplicative symmetric reductions*) only work because the *synchronous* and also the *full synchronous* π-calculus are maybe *too* liberal in their structural congruence, allowing the permuting of (unguarded) prefixes in several cases that are not usually allowed in concurrency calculi.

Although these are rather serious problems, this approach is the best already found relating a logic to concurrency in the style of the Curry-Howard isomorphism. It has two important qualities. First, the logic was not modified—no extra rule or extra connective was added to the system in order to represent concurrency features; therefore, no logical atrocity was committed. Instead, the concurrency calculus was slightly modified, which is more acceptable. Second, even though the results were not completely satisfactory, Abramsky's insights (the 'proofs as processes' paradigm) make sense and linear logic is more adequate to represent

concurrency than any of the other logics previously used. For these reasons, this approach can serve as a starting point for future works relating concurrency to logical systems.

4.6 Conclusion

There are several works in the literature relating logic to concurrency. The objectives of these works may be, among other things, to develop a typing discipline for concurrent processes, or to prove normal form theorems for models of concurrency (see Chapter 3). These works have appeared in part as a consequence of the success of the works relating intuitionistic logic to sequential and functional computation such as, for example, the Curry-Howard Isomorphism (CHI) [How80], Martin-Löef's Intuitionistic Type Theory [Mar84] and Labelled Natural Deduction [dG92]. It would be interesting to obtain for concurrency a result as nice as CHI, which establishes a strong relationship between intuitionistic logic proofs and λ-calculus terms and which provides the foundation for one of the most successful applications of formal methods: typing disciplines. Such a result would certainly increase the usability of models of concurrent computation.

At a first moment, it has been tried to achieve results such as this one by using classical logic instead of intuitionistic logic. However, this was not possible since classical logic reduction relation is not confluent (it does not have the Church-Rosser property). Therefore, the next step was to try to relate modal logic and concurrency. This is a very natural idea, since modalities such as necessity and possibility as well as temporal modalities can be used to represent the dynamic

behaviour of processes. Although the first works relating modal logic and concurrency have appeared a long time ago [HM85], it has not yet been possible to obtain results as good as CHI for the relationship between modal logic and concurrency. Recently some interesting works relating modal formalisms (μ-calculus) and models of concurrency have appeared, such as [AD96], which was discussed in this Chapter.

The advent of linear logic [Gir87] has led to a new approach to the interaction logic-concurreny. The idea, called 'proofs as processes', is to adapt the 'propositions as types' paradigm (which has led to CHI) to the concurrency world, by using linear logic proofs and formulas to 'type' concurrent processes. Several works in this approach have appeared [**?**, Abr93, BS94] relating linear logic to algebraic models of concurrency. The most important results of this approach, in our opinion, are presented in Bellin and Scott's work. However, one can not conclude from these results that linear logic is the most adequate logic to handle concurrency features. This happens for two main reasons: (i) the processes that can be typed in these works are (yet) too restricted, and (ii) even for the processes that can be typed the results are not as strong as CHI. Notwithstanding, these works are a good point of departure for future works relating logic to concurrency.

æ

Chapter 5

Using LDS for Concurrency

5.1 Introduction

In this Chapter we will discuss the definition of a Labelled Deductive System (LDS) for concurrency. We will present a preliminary version of a system that intends to represent a logic for concurrency inspired by the Curry-Howard functional interpretation and by Abramsky's 'proofs as processes' paradigm. First, we will discuss LDS's origins (motivated by the proliferation of logics in the 1980's as well as by subsequent discussions on the general structure of logical systems) and present its definition. Secondly, we will discuss Labelled Natural Deduction (LND), an instance of LDS that seems to be adequate for providing a logical presentation of concurrency.

After that, we will talk about some of the features we consider necessary for making up a logical system for concurrency (such as, e.g. a logical interpretation of

concurrency connectives). Following, we present a preliminary definition of such a system for a restricted process algebra. Finally, we conclude by discussing the existing problems of this version as well as the possible ways to extend and to improve our system in order to achieve the aims proposed for it.

5.2 Labelled Deductive Systems

The framework of Labelled Deductive Systems has appeared as a tentative to solving the problem of *proliferation of logics* in the 80's. Due to the new applications that became possible through the use of computers, it became necessary to develop several logical systems for areas such as linguistics, philosophy and computer science. These logical systems that have appeared (and continue to appear) are usually distinguished from each other by considerations performed in the 'meta-level'; that is, the distinctive features do not make part of the object language of the logic, but are rather observations such as "how many times was this formula used in order to obtain this other formula?" or "what is the degree of probability of this formula?" Aspects such as resource use and fuzziness were not adequately represented in logical systems and had to be described using meta-level considerations.

Therefore, Gabbay's intention when he introduced LDS [Gab95] was to try to incorporate into the object level everything that was taken into account in the meta-level of these new logical systems. In this way, it was first necessary to know what is a logical system (i.e. to have a definition of such a kind of system that would incorporate these new logical systems) and how the information described

in the meta-level could be taken into account in the definition of consequence relations and other features of these systems. That is, the idea was to have *all* the information represented in the object-level, leaving nothing to be handled in the meta-level.

As a result of these discussions about the nature of a logical system [Gab95], Gabbay developed the notion of LDS. The LDS perspective is a mode of presentation of logical systems where meta-level features can reside side by side with object level features. It is a powerful and flexible tool for the definition of logical systems. It is also a general and unifying framework, since several different logical systems can be presented as an LDS; in Gabbay's book [Gab95], for example, he shows how to present concatenation logic, modal logic and many other logics.

The LDS perspective can be differentiated from other forms of presentation of logical systems mainly due to its notion of declarative unit. In LDS, the declarative unit is the pair $t : \mathbf{A}$, where t is a *label* and $\mathbf{A}$ is a *formula*. The label is an extra information about the formula (represented in the meta-level in usual logical systems) and may denote different things depending on the particular application, including the fuzzy reliability value of the formula, a proof of the formula, etc. "The label is meant to carry information which may be of a less declarative nature than that carried by the formulas. The introduction of such an extra dimension was motivated by the need to cope with the demands of computer science applications." [dG94].

Since in the LDS methodology labels are part of the declarative unit[1], one needs

[1]And according to Gabbay [Gab95], "this is not the same as the occasional use of labelling with some specific purpose in mind", because "we are claiming that the notion of a logic is an

to define a notion of a *labelled consequence relation*. In LDS, the consequence relation is defined by using rules on both formulas and their labels. The logic and the logical steps shall remain simple, because there is a separate but harmonious calculus in the labels that is in charge of handling the complexity. Notwithstanding, according to de Queiroz and Gabbay [Gd92], "The consequence notion for labelled deduction is essentially the same as that of any logic: given the assumptions, does a conclusion follow?" A definition of LDS is presented below:

Definition 5.2.1 (Prototype algebraic *LDS* [Gab95]) Let $\mathcal{A}$ be a first order language with a set of terms (which will be the atomic labels), some function symbols (which generate more labels from the atomic labels) and some predicate symbols (which give additional structure to the labels). A *diagram* of labels is a set $\mathcal{D}$ containing the elements generated from $\mathcal{A}$ by the function symbols together with formulas of the form $\pm\mathbf{R}(t_1, \ldots, t_k)$, where $t_i \in \mathcal{D}$ and $\mathbf{R}$ is a predicate symbol from the algebra.

Let $\mathcal{L}$ be a predicate language with connectives $\sharp_1, \ldots, \sharp_n$ of various arities, with quantifiers and with the same set of atomic terms $\mathcal{A}$ as the algebra.

We define the notions of a declarative unit, a database and a label as follows:

- An atomic label is any $t \in \mathcal{A}$. A label is any term generated from the atomic labels by the symbols $f_1, \ldots, f_m$.
- A formula is any formula of $\mathcal{L}$.
- A declarative unit is a pair $t : \mathbf{A}$ where t is a label and $\mathbf{A}$ is a formula.

LDS".

- A database is either a declarative unit ot has the form $\langle \mathcal{D}, f, d, U \rangle$, where $\mathcal{D}$ is a finite diagram of labels, $d \in \mathcal{D}$ is the distinguished label, and f is a function associating with each label t in $\mathcal{D}$ either a database or a finite set of formulas and U is the set of all terms.

□

Any logical system can be formulated in LDS. However, one does not intend to substitute in practice the original formulation of a logical system for its LDS formulation [Gab95]. The LDS presentation serves to compare logical systems, analyse them as well as to make it easier to obtain extensions and/or restrictions to logical systems, since everything becomes more explicit and manageable using two levels: a calculus on the formulas and another calculus on the labels.

LDS has found several applications in linguistics, philosophy and computer science [dG92]. One of the most important 'applications' of LDS, one that shows clearly how it is important and useful to have two levels in a logical system, is the Labelled Natural Deduction. Besides being an instance of LDS, LND can also be seen as a framework for studying the mathematical foundations of LDS, since the Curry-Howard interpretation (on which LND is based) can be viewed as a labelling scheme for intuitionistic well-formed formulas.

5.3 Labelled Natural Deduction

As an instance of LDS, LND can also be used in order to present different logical systems. LND is based on an extension of the Curry-Howard functional interpreta-

tion. In the Curry-Howard functional interpretation, each formula is accompanied by a functional term such that (i) the term represents a *construction* of the formula, that is, a *proof* of the formula and (ii) the formula can be seen as a type of the term (formulae-as-types paradigm). Therefore, a formula (type) is valid if and only if there is a term that inhabits that type (such as, e.g. formula $\mathbf{A} \Rightarrow \mathbf{A}$ is valid because $\lambda x.x$ inhabits type $\mathbf{A} \Rightarrow \mathbf{A}$).

The functional interpretation originated in 1934, when Curry demonstrated that there was a close correspondence between the axioms of intuitionistic implicational logic and the type schemes of the so-called 'combinators' of Combinatory Logic (B, C, K, etc.). Later, Howard established an isomorphism between intuitionistic logic and a functional calculus (λ-calculus). The so-called Curry-Howard functional interpretation was conceived as a 'notion of construction' for intuitionistic logic. But it was Tait's intensional semantics based on convertibility that allowed flexibility in obtaining a functional interpretation "for many other logics including those which do not abide by the tenets of intuitionism" [Gd92]. And actually, according to Gabbay and de Queiroz [Gd92] it was in Frege that the two facets of formal logic (functional calculus on terms and logical calculus on formulas) were first put together in a system (the system that formalizes arithmetics, Grundgesetze [Fre93], alongside the system of concept writing, Begriffsschrift [Fre79]).

In LND the functional calculus is in charge of handling names, referents and so on (information contained on the labels), as well as of recording the steps performed in the proof, whilst the logical side takes care of the formulas. The two sides are separate but harmonious; there is no interference between the calculi. With this apparatus, one can provide an extension of the functional interpretation

that enables one to account for several different connectives (implication, universal quantification, necessity) of many logics (resource logics and even classical logic, among others).

In [Gd92], it was shown how to extend the functional interpretation (using LND) in order to represent *implication* in the so-called resource logics (linear, relevant and etc.). They demonstrated that "just by working with side conditions on the rule of *assertability conditions* for the connective representing implication ('⇒') one can characterise those 'resource' logics" and that only the 'improper' inference rules leave 'room for manouevre' as to how a particular logic can be obtained. We can see this more clearly in the following extract from [Gd92]:

> The rule of ⇒-*introduction* is classified as an 'improper' inference rule, to use a terminology from Prawitz [1965]. Now, the so-called improper rules leave room for manoeuvre as to how a particular logic can be obtained just by imposing conditions on the discharge of assumptions discipline one is adopting (linear, relevant, ticket entailment, intuitionistic, classical, etc.). The side conditions can be 'naturally' imposed, given that a degree of 'vagueness' is introduced by the presentation of those improper inference rules, such as the rule of ⇒-*introduction*:
>
> $$\frac{\begin{array}{c}[\mathbf{A}]\\ \mathbf{B}\end{array}}{\mathbf{A} \Rightarrow \mathbf{B}}$$
>
> which says: starting from assumption '**A**', and arriving at '**B**' via an

unspecified number of steps, one can discharge the assumption and conclude that '**A**' implies '**B**'.

Note that one might (as some authors do) insert an explicit indication between the assumption '[**A**]' and the premise of the rule, namely '**B**', such as e.g. the three vertical dots, making the rule look like:

$$\frac{\begin{array}{c}[\mathbf{A}]\\ \vdots\\ \mathbf{B}\end{array}}{\mathbf{A} \Rightarrow \mathbf{B}}$$

drawing attention to the element of vagueness. The more specific we wish to be about what the three dots ought to mean, the more precise we will be with respect to what kind of implication we shall be dealing with.

Regarding the definition of implication, one can also notice that different logical implications can be presented as different sets of Hilbert style axioms. For example, intuitionistic implication can be defined by the set of axioms in Table 5.1 whilst linear implication is defined only by the axioms in the left side of that table. In LND, a connective is not defined by a set of axioms but rather by its introduction, elimination and reduction rules. Since only the introduction rules for implication can be characterised as improper, in order to define linear implication, for example, we must provide a definition for $\Rightarrow$-introduction such that the axioms in the *right* side of Table 5.1 cannot be derived. How can we do that? In LND, we can do that just by working on the assertability conditions of these rules, restricting λ-

$$\mathbf{A} \Rightarrow \mathbf{A}$$

$$(\mathbf{A} \Rightarrow \mathbf{B}) \Rightarrow ((\mathbf{C} \Rightarrow \mathbf{A}) \Rightarrow (\mathbf{C} \Rightarrow \mathbf{B})) \qquad (\mathbf{A} \Rightarrow (\mathbf{B} \Rightarrow \mathbf{C})) \Rightarrow ((\mathbf{A} \multimap \mathbf{B}) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{C}))$$

$$(\mathbf{A} \Rightarrow \mathbf{B}) \Rightarrow ((\mathbf{B} \Rightarrow \mathbf{C}) \Rightarrow (\mathbf{A} \Rightarrow \mathbf{C})) \qquad \mathbf{A} \Rightarrow (\mathbf{B} \Rightarrow \mathbf{A})$$

$$(\mathbf{A} \Rightarrow (\mathbf{B} \Rightarrow \mathbf{C})) \Rightarrow (\mathbf{B} \Rightarrow (\mathbf{A} \Rightarrow \mathbf{C})) \qquad \mathcal{F} \Rightarrow \mathbf{A}$$

Table 5.1: Hilbert-style axioms for intuitionistic and linear implication

abstractions in the labels, thus also restricting the supply of definable terms and getting fewer theorems and axioms.

For example, axiom $\mathbf{A} \Rightarrow (\mathbf{B} \Rightarrow \mathbf{A})$ is not valid for relevant implication. Therefore, when defining relevant implication in LND, that axiom must not be valid. The derivation of this axiom in Figure 5.1 can be overruled by imposing a restriction on the formation of λ-terms on the labels, the restriction that "there must be at least one free occurrence of the variable in the term on which the abstraction is operating" [Gd92]. There, "the discharge/abstraction of the assumption '$[y : \mathbf{B}]$' is made over the expression 'x' in '$\lambda y.x$', which prevents it from being considered 'relevant' ". In [Gd92] it is also shown how to extend the set of definable types (and of theorems) obtaining implications *stronger* than intuitionistic implication.

5.4 Towards a LDS for Concurrency

Here we discuss the development of a *Labelled Deductive System* (LDS) for concurrency as a *logical approach* to the resolution of problems concerning the formal

$$\dfrac{\dfrac{\dfrac{[y : \mathbf{B}]}{x : \mathbf{A}}}{\lambda y.x : \mathbf{B} \Rightarrow \mathbf{A}}}{\lambda x.\lambda y.x : \mathbf{A} \Rightarrow (\mathbf{B} \Rightarrow \mathbf{A})}$$

Figure 5.1: An invalid axiom for relevant implication

specification of concurrent systems. *Concurrency,* an important subject in computer science, still lacks a better foundation for some of its aspects. Its main problems, in our point of view, are the lack of a typing discipline for processes and the lack of a normal form theorem for processes (see Chapter 3). We think that these problems can be solved (at least partially) by the utilization of logic to give better foundations to concurrency. With the purpose of presenting solutions to these problems we attempted to develop an LDS for concurrency.

There are several works in the literature that offer a logical approach to the solution of problems in concurrency [Abr93, BS94]. Our approach in this paper is similar to the approach in [Abr93] and in [BS94]. These works tried to establish a relationship between *linear logic* (a new logical system presented by Girard in [Gir87]) and the π*-calculus* (a concurrency calculus developed by Milner [Mil93] which allows the description of mobile processes) as strong as the Curry-Howard isomorphism [How80]. Here we have the following *aims*: (i) to develop a *typing discipline* for a (maybe restricted) algebraic process calculus; (ii) to find an appropriate formulation of a *normal form theorem* for processes of this calculus, based on adequate proofs (as processes) equivalences; and (iii) to define notions of equivalence and congruence between these processes. Since we are using LDS instead

of any particular logical system such as linear logic, we expect to be able to keep the logic for concurrency as close as possible to classical logic. The intention is to be able to handle concurrency features in LDS's meta-level, i.e. in the labels.

How can classical connectives represent concurrency features? It is a fact that classical connectives are at a higher level of abstraction than linear connectives, because classical logic is mostly concerned with the formalization of the kind of reasoning used in mathematics. Therefore, the use of formulas is not counted since it does not matter for proofs in mathematics. But, by using LDS one expects to be able to put all computational features in the labels, therefore making it possible to type concurrent processes with classical connectives.

Thus, a LDS for concurrency will be a logical system where the calculus on the labels is a process algebra (such as CCS and π-calculus) and in the formulas we will have types of concurrent processes. That is, in the declarative unit '$t : \mathbf{A}$', t will be a term of a process algebra and $\mathbf{A}$ will be a formula that shall specify some features of this process term: its type. Our problem is to find a logic that can accompany the process terms in the labels (this logic would function as a typing discipline for the concurrent algebra) in the same way that intuitionistic logic is accompanied by (and is a typing discipline for) the λ-calculus in Curry-Howard functional interpretation.

5.5 Analysis of Constructors

Having in mind this aim of finding a 'logic for concurrency', we first have to choose a specific process algebra (to be our algebra in the labels) and then try to provide

a *logical interpretation* to its rules and constructors. The logical interpretation of a concurrency constructor consists of finding one (or more) logical connectives that can be used to represent (or that serve as types of) that constructor. There must be an association between the introduction, elimination and reduction rules of that constructor and the respective rules of the logical connective. For example, in the functional interpretation of logical connectives there can be found some similarity between the rules for implication and the rules for abstraction in the λ-calculus. Therefore, in many cases abstraction (see [Gd92]) has as type a logical expression where the main connective is implication (or, in our terms, implication is the logical interpretation of abstraction).

It is easy to notice that the direction we are going is the opposite to the one followed by the works that perform a functional interpretation of logical connectives; i.e. instead of finding a (functional or concurrent) calculus that can adapt to a given logic, we try to find a logic that can adapt to a specific concurrency calculus. It is much more difficult to obtain such a result even because Howard's result in [How80] was based on several other results (Frege, Heyting, Curry, Tait) and tried to establish only a notion of construction for intuitionistic logic; the typing discipline for λ-calculus was a side result.

Regarding concurrency, the existing models of concurrency are relatively new and were not designed in order to have a logic foundation, neither to be amenable to a typing discipline (in most cases it was not possible to find a good typing discipline for them), since even their development was already quite a challenge. They were made only to work in practice, i.e. to serve as a good theoretical foundation to concurrent programming languages. But, since some models (such as, e.g. CCS

and CSP) have achieved this objective quite reasonably, don't they have some kind of logical basis behind them? If there is such a logical basis, that is what we intend to find.

In order to obtain a logical interpretation of concurrency constructors we have to analyse and compare them to logical connectives defined in some logical presentation system. We regard LND as the most adequate system for achieving this aim. This occurs because of two things. First, plain natural deduction is not able to make distinctions between certain connectives, unless by observations performed in meta-level. For example, how could one differentiate intuitionistic implication from linear implication? [Gd92].

Secondly, in Gentzen's sequent calculus the left and right connective rules are more complicated and have less intuitive meaning than natural deduction rules, since these rules have to take into account contexts, order of formulas in the sequents, mathematical structure underlying sequents and so on. LND, however, allows different connectives of distinct logics to be represented by restrictions in the labels that accompany the formulas [Gd92]. The intuitive meaning of the natural deduction rules is not lost (because there is a clear separation between logical concern on the logical side and other features on the labels side) and the local control existing in sequent calculus is maintained.

Besides that, in sequent calculus all reductions are related to only one rule, the Cut rule. The reductions either move the Cut or eliminate it. In natural deduction (and also in LND) the reductions involve directly introduction and elimination rules of each connective, therefore making the study of these rules simpler.

$$\frac{P : \mathbf{T} \qquad Q : \mathbf{U}}{P \parallel Q : \mathbf{T} \wedge \mathbf{U} \wedge \mathbf{X}}$$

Figure 5.2: Parallel composition in an LDS for concurrency

5.5.1 Parallel Composition

The most important concurrency constructor is parallel composition. Without it, a process algebra can not even be considered concurrent. The parallel composition of two given processes is obtained from (the existence of) two processes simply by putting them to work together concurrently. No restriction is made regarding the link between the two processes or the synchrony between them, at least in an asynchronous calculus. When two processes are composed in parallel, they may proceed independently one of the other or even communicate between themselves. It is important to notice that once this constructor is introduced, it only disappears if and when at least one of the processes terminates completely its execution, becoming the Zero process. And that demands a reduction that is in fact a rewrite, since it does not represent a computational step: $A \parallel 0 \longrightarrow A$.

Due to these features, the classical logic connective that most resembles parallel composition is conjunction. However, it is necessary more than conjunction to type parallel composition. We have not yet found what more is necessary to type it. See Figure 5.2, where $\mathbf{X}$ represents the extra features that are needed in order to represent parallel composition (and that probably are going to be represented in the labels).

5.5.2 Nondeterminism

Another very important concurrency constructor, present in most process algebras, is the nondeterminism operator (+). It is so important that is some calculi it is considered to be more primitive than parallel composition. For example, if we consider (in a simplifying fashion) that in a concurrent system there is only one observer that can see only one action at a time (see discussion on interleaving assumption in Chapter 3), then every process written using parallel composition can be rewritten using + instead of $\|$. In this case, e.g a process $a.0 \parallel b.0$ would be rewritten to $a.b.0 + b.a.0$.

A process $P + Q$ is constructed from (the existence of) two processes P *and* Q. This is a conjunctive feature, but the process $P + Q$ will actually behave as P *or* as Q (a disjunctive feature that we regard as stronger). In practice, this process will behave as P or Q depending on particular conditions such as, e.g. an input received from the environment (ex.: $(a.P + b.Q) \parallel \overline{a}.0 \longrightarrow P \parallel 0$). In this way, the logical connective chosen to represent nondeterminism is disjunction, putting in the labels the demands made by the conjunctive features of +.

Nondeterminism is manifest when a process asks for one action that any of the two (or more) processes joined by + could attend (ex.: $(a.P + a.Q) \parallel \overline{a}.0$ can be reduced either to $P \parallel 0$ or to $Q \parallel 0$). Then, an internal choice is performed deciding which of the processes will be used. One can see that this is different from intuitionistic disjunction (see [dG92]), where one already knows, by looking at the expression in the label (either `inl` or `inr`), which of the two formulas is going to be used by the `case` destructor.

In some calculi, such as in [Hen88], there is another kind of nondeterminism operator: internal nondeterminism ($\oplus$). In this operator, the choice of P or Q is always made internally. It is also used, along with (external) nondeterminism, to represent parallel composition in that system. As we have seen in subsection 3.4.1, it has appeared because one needed a counterpart in the process algebra to a mathematical structure that appeared in the denotational semantics. It has also proved to be intuitively valid. Milner, for example, represents $\oplus$ in the π-calculus as derivable from $+$ and τ[2]: $P \oplus Q \equiv \tau.P + \tau.Q$.

5.5.3 Action Prefixing

Action prefixing has appeared in Milner's CCS in order that only one kind of process composition be necessary in the calculus, i.e. to prevent the existence of two types of process composition: parallel and sequential composition. This is a valid justification, since the calculus becomes more basic in this way. In action prefixing, one does not say that a process is first executed and after that another process is executed; here, given an action a and a process P, we build the process $a.P$ (action a prefixed to P) such that this process first performs a and then becomes process P. From this, several sequential processes can be constructed (an algebra which has only action prefixing can not be regarded as concurrent) and the sequential composition of processes can be derived from parallel composition. Suppose that ; is sequential composition, that there are two processes $P = a.b.c.0$ and $Q = d.e.f.0$, and that g is a special synchronizing action. Then, it is possible

[2]And τ is defined in the following way: a process $\tau.P$ is $(\upsilon a)(a.P \parallel \overline{a}.0))$.

to define $P;Q$ as being the same as $(\upsilon g)(P' \parallel Q')$, where $P' = a.b.c.g.0$ and $Q' = g.d.e.f.0$. Action g guarantees the sequentialization of the two processes.

We think that the logical connective with more resemblance in meaning to action prefixing is implication. But what kind of implication? A kind of linear implication, since a port name can appear and is used exactly once when it prefixes a process. Implication reasonably represents action prefixing because $\mathbf{A} \Rightarrow \mathbf{B}$ can be read as 'from **A** follows **B**', that is, one needs to have **A** in order to have **B**. In a similar way, given a process $a.P$, first a (an atomic action) must happen and after that P (a possibly complicated process) is available. This implication is regarded as linear because a must occur (happen) exactly once so that P becomes available. The biggest difference between the meanings of implication and action prefixing is that in the implication introduction many things can occur between the first assumption ('[**A**]') and the conclusion immediately before the discarding of the assumption ('**B**'). On the other hand, in the construction of a process with action prefixing, following the presentation of the port one goes directly to the constructed process. We do not have a justification, however, for the fact that the port is presented as an assumption (with square brackets around it) in rule $\Rightarrow I$ (see Section 5.6).

5.5.4 Zero Process

When presenting some modal connectives in LND, Gabbay and de Queiroz introduced a formula called $\mathcal{U}$ in order to logically represent the universe of all possible worlds. Here, in a similar way, we introduce a formula to represent a unary con-

structor (also considered to be a constant) present in most process algebras: the 0 process, used to get constructions started. The $\mathcal{Z}$ formula is created only to type this process and does not have a special meaning such as $\mathcal{U}$ in modal logic or as $\mathcal{F}$ (used to represent a formula that is always false in intuitionistic logic such as, e.g. $\mathbf{A} \wedge \neg\mathbf{A}$). Notwithstanding, $\mathcal{Z}$ is essential because without it there is no construction of types and because it represents the end of the execution of a process. In this system, however, we did not include any rule to eliminate $\mathcal{Z}$, such as $\mathcal{Z} \wedge \mathcal{Z} \equiv \mathcal{Z}$. This treatment is left for the reduction rules to be developed in the future.

5.6 Definition of the LDS for Concurrency

In this preliminary version of a LDS for concurrency, the language $\mathcal{L}$ is composed of the following connectives $\wedge$ and $\Rightarrow$, plus capital letters to represent propositions. There is one special proposition, $\mathcal{Z}$, that is the type of the Zero process and that is always valid. There are no quantifiers nor other connectives in this version of the system.

The algebra on the labels is based on a *restricted* version of CCS (which we call RCCS—a version of CCS with only three constructions to describe agents: Zero, Action Prefixing and Parallel Composition). The descriptions of concurrent processes (terms of an algebraic calculus) are written according to RCCS rules and $\mathcal{A}$ (from Definition 5.2.1) is RCCS (using an infix notation for the function symbols mentioned in Gabbay's definition) plus the following functional symbols necessary to represent elimination rules: `fst`, `snd`, `APP`).

Here, the **labels** are *descriptions* of concurrent processes, and the **formulas** are *types* of these processes. Types are high level specifications of processes and, according to the 'propositions-as-types' paradigm, they can be represented by formulas in a logical system. In this system, RCCS's *ports* have atomic formulas as their types whilst *processes* are typed by composite formulas. Complementary ports are assigned to the *same* formula as type. We do not use negation for this purpose, as has been done in [Abr93] and [BS94], because in LDS the label is an 'integrant' part of the declarative unit and of the logic and, therefore, it is not necessary to make the distinction between complementary ports also in the formulas. A justification for this choice is the $\Rightarrow E$ rule (modus ponens), which would not be possible without this decision.

The system is a kind of natural deduction which carries context information in the labels. This 'bookeeping' is necessary in order to represent many things in concurrency, including communication. For example, two processes may communicate in a parallel composition where a third process (the context) remains unaltered: $P \parallel Q \parallel R \longrightarrow P' \parallel Q' \parallel R$. Each proof in the system is represented in a tree-like form, with only one conclusion. This is OK in concurrency formalisms such as CCS because all processes in a concurrent system must be joined by parallel composition. Thus there is no need for multiple conclusions. There is a problem with this tree-like form regarding communication (we see this in next subsection) but it can be overcome. The general form of the system is therefore very similar to the *Labelled Natural Deduction* (LND) system [dG92], an instance of LDS.

As in LND, for each logical connective there are three types of rules: *introduction*, *elimination* and *reduction* rules. The introduction rules are responsible for the

construction of processes, the elimination rules represent actions being performed (including the ones involved in communication) whilst reduction rules represent normalization of processes, i.e. the rewriting of processes after communication. According to Gabbay and de Queiroz in [Gd92], the reduction rules are the ones that define the meaning of a logical connective and of computation constructors.

The system has rules for two logical connectives ($\Rightarrow$ and $\wedge$) and for a new formula $\mathcal{Z}$ that still lacks a good logical explanation (in concurrency the process 0, whose type is $\mathcal{Z}$, is used to get constructions started). The association between logical connectives and concurrency constructors was done based on an analysis of the similarities and differences between them. Here are the rules for the connectives and their explanation:

1. $\mathcal{Z}$ formula - The $\mathcal{Z}$ formula is a special formula that is always valid. It does not depend on any assumption and it can never be eliminated. It is the type of the 0 process, which is used in order to get process constructions started. $\mathcal{Z}$ formula only rule is its introdution rule:

$$\frac{}{0 : \mathcal{Z}}\mathcal{Z}I$$

2. Implication ($\Rightarrow$) - Implication uses the same symbol of classical and intuitionistic implication ($\Rightarrow$) but, due to the information on the labels, it behaves differently. The first rule is implication introduction ($\Rightarrow I$). Its form is very similar to the LND rule for implication and the explanation is the following: if one assumes an atomic formula ($\mathbf{A}$) and proves a formula ($\mathbf{T}$), then one can form the formula $\mathbf{A} \Rightarrow \mathbf{T}$. $\mathbf{A}$ is the type of a RCCS port (such as a or $\overline{a}$) and $\mathbf{T}$ is the type of a process. A process is constructed by the introduction rule

using action prefixing (in this case $\overline{a}.P$, a process that can perform action a and then becomes P) and has $\mathbf{A} \Rightarrow \mathbf{T}$ as type:

$$\frac{\begin{array}{c}[\overline{a} : \mathbf{A}] \\ \\ \end{array} \quad \begin{array}{c} \vdots \\ P : \mathbf{T} \end{array}}{\overline{a}.P : \mathbf{A} \Rightarrow \mathbf{T}} \Rightarrow I$$

The elimination rule is *modus ponens*, but also taking into account the information about processes and ports in the labels. The following computation is depicted in the rule: if you have a process $\overline{a}.P$ of type $\mathbf{A} \Rightarrow \mathbf{T}$, then this process can communicate with an action a (complementary to the action prefixed to the other process) of type $\mathbf{A}$. The resulting process ($\mathtt{APP}(\overline{a}.P, a)$) is equivalent to P after performing τ, and its type is $\mathbf{T}$.

$$\frac{\begin{array}{c} \vdots \\ \overline{a}.P : \mathbf{A} \Rightarrow \mathbf{T} \end{array} \quad \begin{array}{c} \vdots \\ a : \mathbf{A} \end{array}}{\mathtt{APP}(\overline{a}.P, a) : \mathbf{T}} \Rightarrow E$$

3. Conjunction ($\wedge$) - Introduction rule for conjunction represents the formation of the type (formula) $\mathbf{T} \wedge \mathbf{U}$ from the derivations of the types (formulas) $\mathbf{T}$ and $\mathbf{U}$. In the labels, the construction of the parallel composition of two processes accompanies this formation of the type:

$$\frac{\begin{array}{c} \vdots \\ P : \mathbf{T} \end{array} \quad \begin{array}{c} \vdots \\ Q : \mathbf{U} \end{array}}{P \parallel Q : \mathbf{T} \wedge \mathbf{U}} \wedge I$$

The elimination rules show the decomposition of a formula whose main connective is $\wedge$. Operators called 'first' (`fst`) and 'second' (`snd`) are used in

these rules in order to select the first (second) conjunct in the logical side (respectively), in a similar way to what happens in LND conjunction rules [dG92]. This decomposition must take into account the fact that one cannot *discard* a process from a parallel composition of processes. This is going to be represented only in the reduction rule that we are going to see in the next subsection:

$$\frac{P \parallel Q : \mathbf{T} \wedge \mathbf{U}}{\mathtt{fst}(P \parallel Q) : \mathbf{T}} \wedge E_1 \quad \frac{P \parallel Q : \mathbf{T} \wedge \mathbf{U}}{\mathtt{snd}(P \parallel Q) : \mathbf{U}} \wedge E_2$$

The rules for the logical connectives are very similar, in the logical side, to the rules of LND for the same connectives of intuitionistic logic. However, the information recorded in the labels is used to restrict the set of provable formulas according to concurrency features. That is, in order to have a valid deduction one must obbey concurrency laws (such as the one that establishes that only complementary ports can communicate, depicted in $\Rightarrow E$), as well as logical laws.

5.6.1 Representation of Communication

The intention here is to discuss the representation of communication in an LDS for concurrency. Communication in RCCS is the interaction between exactly two processes. An exchange of messages between these two processes, which are composed in parallel (maybe with other processes), occurs along communication. This is what happens: one process *sends* a message at a given port (such as a—this is represented in the calculus by the process performing action $\overline{a}$) whilst the other process *receives* this message in the same port (represented by the process doing action a). After that, the processes resume their normal operation and other

communications may take place.

Communication is a very important issue in concurrency mainly because it gives the meaning of the most important concurrency constructor: parallel composition. A different definition of communication would imply in another kind of parallel composition. In RCCS, communication is represented by the following rule in the reduction relation of the calculus:

$$\overline{a}.P \parallel a.Q \parallel \ldots \xrightarrow{\tau} P \parallel Q \parallel \ldots$$

From this rule we devised the following *proof transformation* that represents communication in our LDS for concurrency:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\overline{a} : \mathbf{A}]}{\dfrac{\overline{0 : \mathcal{Z}}}{\overline{a}.0 : \mathbf{A} \Rightarrow \mathcal{Z}} \Rightarrow I} \quad \dfrac{\dfrac{[a : \mathbf{A}]}{\dfrac{\overline{0 : \mathcal{Z}}}{a.0 : \mathbf{A} \Rightarrow \mathcal{Z}} \Rightarrow I}}{\overline{a}.0 \parallel a.0 : (\mathbf{A} \Rightarrow \mathcal{Z}) \wedge (\mathbf{A} \Rightarrow \mathcal{Z})} \wedge I}{\dfrac{a : \mathbf{A} \quad \mathtt{fst}(\overline{a}.0 \parallel a.0) : \mathbf{A} \Rightarrow \mathcal{Z}}{\mathtt{APP}(\mathtt{fst}(\overline{a}.0 \parallel a.0), a) : \mathcal{Z}} \Rightarrow E \quad \dfrac{\overline{a} : \mathbf{A} \quad \mathtt{snd}(\overline{a}.0 \parallel a.0) : \mathbf{A} \Rightarrow \mathcal{Z}}{\mathtt{APP}(\mathtt{snd}(\overline{a}.0 \parallel a.0), \overline{a}) : \mathcal{Z}} \Rightarrow E}{\mathtt{APP}(\mathtt{fst}(\overline{a}.0 \parallel a.0), a) \parallel \mathtt{APP}(\mathtt{snd}(\overline{a}.0 \parallel a.0), \overline{a}) : \mathcal{Z} \wedge \mathcal{Z}} \wedge I} \wedge E$$

$$\Downarrow$$

$$\dfrac{\overline{0 : \mathcal{Z}} \quad \overline{0 : \mathcal{Z}}}{0 \parallel 0 : \mathcal{Z} \wedge \mathcal{Z}} \wedge I$$

This first part of this transformation (above the $\Downarrow$ symbol) allows us to identify *five* steps in the process of communication:

1. the construction of the processes which are going to participate in the parallel composition, among them the two which are going to communicate. In this case, $\overline{a}.0$ and $a.0$, constructed by the $\Rightarrow I$ rules;

2. the parallel composition of those processes. Here, the process that results from the parallel composition is $\overline{a}.0 \parallel a.0$, composed by the first $\wedge I$ rule;

3. the 'choice' of the processes in the parallel composition which are going to communicate ($\wedge E$ rule). This choice is important when there are more than two processes composed in parallel (which is not the case here);

4. the 'application' of the actions to the processes chosen ($\Rightarrow E$ rules). These actions ($a : \mathbf{A}$ and $\overline{a} : \mathbf{A}$ in the rule above) are the actions performed in the communication;

5. the 'reconstruction' of the parallel composition after communication has happened (second $\wedge I$ rule). Here we must have in mind that the processes that were not chosen in Step 3 (i.e. that did not participate actively in the communication) must remain unchanged and be 'restored' to the parallel composition of processes (e.g. in $P \parallel Q \parallel R \longrightarrow P' \parallel Q' \parallel R$ and R remains unaltered since there is no broadcasting). This is a kind of resource awareness that can be checked in the labels.

The second part of this proof transformation (below the $\Downarrow$ symbol) shows the proof of the process that results from the computation in the first part, that is, the process that remains after the communication. It shows the construction of this process (in this case $0 \parallel 0$) as if it were not the result of a communication, but as a process constructed from scratch.

This representation of communication has both nice features and problems. One of the interesting features is the separation of the processes that actually communicate

from the processes that do not take part in the communication (Step 3). Also, the possibility of taking into account the use of resources in the labels (Steps 3 to 5). However, two important problems are the lack of a better logical explanation for the appearance (in the proof) of the actions in Step 4 and the lack of a logical account for the *silent action* (τ), which in [Abr93] and [BS94] was considered to be the elimination of a cut in linear logic cut-elimination process.

5.7 Analysis of the Results

This work has failures. Although we have identified similarities between concurrency constructors and logical connectives (others had already done this [BS94]), the obtained rules are not yet expressive enough, that is, they do not describe accurately what we want to describe: some features of concurrency constructors (see Chapter 3). Also, the only reduction rule presented (the communication rule) is too complicated (has several parts) and only works for the very restricted RCCS calculus. RCCS does not even have the nondeterminism operator, as well as other constructors that were out of our analysis above such as, e.g. replication, recursivity, and action prefixing with name or parameter passing.

However, this work is an attempt at what seems to be a difficult task: using a logic as close as possible to classical logic (in the LDS framework) in order to represent concurrency. By manipulating concurrency features mostly in the labels we intend to keep the logic as simple as possible. Also, we expect to be able to extend the resulting system (due to LDS features) in several ways (as we can see in subsection 5.8). Therefore, in the future we will try to obtain a more concrete

result for a restricted calculus such as RCCS and, after that, extend our results to concrete applications in order to find out practical applications of these results.

5.8 Extensions

Once we have a LDS/LND for concurrency it will be possible to extend it in order to achieve several interesting results. For example, we could add another label representing of the probability of a process. In this way, a process such as $P + Q$ would become, for example, $0.9 : P + 0.1 : Q$, describing that the occurrence P is more probable than the occurrence of Q.

Another possibility would be to use other relations between formulas (besides the consequence relation - such as abduction, etc. [Gab95]) in order to obtain new relationships between processes. One could also link processes to databases (processes that update databases - for example, a process that performs the intersection of two lists of records from differents parts of a database) by using LDS's databases.

More importantly, one could type processes with connectives from different logics (intuitionistic, relevant or linear implication, for example) in the same system, this difference between connectives being established by restrictions on the rules, especially in the labels. In this context, it would also be possible to make small changes in already defined rules and obtain slightly different calculi (e.g. obtaining a synchronous calculus from an asynchronous one). Besides that, it would be possible to identify similarities and distinctions between calculi that have different historical origins (and backgrounds), as Gabbay has done with logical systems using LDS.

5.9 Conclusion

In this Chapter we have discussed the possibility of using Gabbay's LDS in order to provide a *logical representation* of models of concurrent behaviour. That is, we intended to use LDS in order to provide a formulation of a logical system (based in the LDS framework) to give a foundation for a model of concurrency. This work is based in the Curry-Howard functional interpretation (which has provided a logical foundation based on intuitionistic logic to functional programming) and also in the recent 'proofs as processes' paradigm [Abr94a] (which we have seen in Chapter 4), the origin of works (e.g. [BS94]) relating linear logic to π-calculus.

First, we have discussed the origins, motivation and features of LDS. LDS is a general framework for the definition of logical systems. An instance of LDS, LND was also studied, since it serves as a mathematical foundation to LDS and it also allows a generalization of the results in Curry-Howard functional interpretation. More importantly, LND serves nicely in order to establish a logical system for concurrency. Following, we have shown what we regard as necessary in order to obtain such a logical system. We attempted to provide a logical interpretation to concurrency constructors of algebraic models such as parallel composition, nondeterminism, action prefixing and zero process.

The main part of this work is yet unfinished. We have here presented only a preliminary definition of a LDS for concurrency, which represents the introduction and elimination rules for three constructors (of a restricted CCS calculus—RCCS) as well as the analysis of the main RCCS reduction rule: communication. Finally, we have noticed the problems that still exist in this definition and discussed some

of the possible extensions and improvements that can be performed in our system.

æ

Chapter 6

Conclusion

In this dissertation we have analysed some of the several approaches used for obtaining a logical foundation to concurrency through a relationship between logical systems and mathematical models of concurrency. These approaches have used different logical systems (such as modal logic and linear logic) aiming at solving some problems of existing models of concurrent behaviour. Our main concern here was to discuss the 'proofs as processes' paradigm, an adaptation of the 'propositions as types' paradigm to the concurrency world that has originated the works relating *linear logic* and algebraic models of concurrency.

Linear logic is a recent development in logic presented by Jean-Yves Girard in 1987 [Gir87]. It has been well accepted by computer science theory research community and several works have appeared studying its proof theory and applications. In Chapter 2 we have discussed some features of linear logic in order to understand the works that relate it to concurrency. Linear logic has several features that are

important for a logic that deals with aspects of computation. For example, we have shown that it is a resource aware logic and a highly expressive system whose connectives are computationally finer than classical logic connectives. In spite of this, linear logic seems to be inadequate to represent computer science applications because it is a very complicated system and also because some of its connectives are still in need of a better intuitive explanation (e.g. $\wp$).

After that, in Chapter 3 we have seen that concurrency is a very important subject in computer science, with many applications. Mathematical models of concurrency are needed in order to describe concurrent systems and also to reason about them. Although the existence of such models represents a relevant improvement on the formal representation of concurrent systems, one can see that many of these models still present some problems, such as, e.g. the lack of an adequate typing discipline for concurrent processes. Here we have discussed some features of models of concurrency in order to discuss the works that try to solve these problems by providing a logical foundation to concurrency. We have also presented two *algebraic* models of concurrent behaviour: CCS and π-calculus.

Abramsky's 'proofs as processes' paradigm is the main approach between the appraoches used for the interaction logic-concurrency discussed in Chapter 4. It was inspired by the 'propositions as types' paradigm, the basis of the successful Curry-Howard funcional interpretation. This interpretation has provided a logical foundation and a typing discipline to functional programming, by relating it to intuitionistic logic. In the works [Abr93, BS94], the 'proofs as processes' paradigm has been used in order to try to solve concurrency problems. Although these works have not been completely successful, if compared to the results obtained

for functional computation and to the objectives proposed by Abramsky [Abr94a], Bellin and Scott's study [BS94] about the correspondence between linear logic and the π-calculus has obtained significative results. For example, it has provided an assignment of π-calculus process terms to linear logic derivations with some interesting properties. However, one can not conclude from those results that linear logic is the most adequate logic to handle concurrency features. This happens for two main reasons: (i) the processes that can be typed in these works are (yet) too restricted, and (ii) even for the processes that can be typed the results are not as strong as those results obtained from the aplication Curry-Howard functional interpretation. Notwithstanding, these works are a good point of departure for future works relating logic to concurrency.

6.1 Future Works

Based on the 'proofs as processes' paradigm and on Curry-Howard functional interpretation, we have presented in Chapter 5 a *preliminary* version of a system whose intention is to represent a logic for concurrency. The idea is to use Gabbay's Labelled Deductive Systems (LDS), a general framework for the presentation of logical systems, in order to provide a logical representation of a model of concurrent behaviour. In that system, the logic shall remain as close as possible to classical logic, since the complication of concurrency features will be handled in the labels. We have discussed the motivations and the possibilities of definition of such a system, but we have not achieved a satisfactory version of it. Therefore, this part of the work is yet unfinished.

In the future, we will continue to study new works that appear relating logic and concurrency, such as [San96]. We also intend to proceed the work of Chapter 5 in order to obtain a LDS system for more expressive concurrency calculi (such as polyadic π-calculus). The practical applications of such a system would be the development of formal methods for constructing concurrent systems as well as the use of the logical system in order to verify properties of concurrent systems' specifications prior to their actual implementation. We plan to test the validity of this logical system with examples taken from practice as well as to reason about properties of real concurrent systems. æ

Bibliography

[AU69] W. Aho and J. D. Ullmann, An Introduction to the Automata Theory, 1969.

[Abr84] S. Abramsky, Reasoning About Concurrent Systems. In *Distributed Systems,* ed. by Chambers & Duce, Academic Press, 1984, 307-319.

[Abr91] S. Abramsky, Tutorial on Linear Logic, handwritten notes, ILPS 91.

[Abr93] S. Abramsky, Computational Interpretations of Linear Logic. *Theoretical Computer Science*, **111** (1993) 3-57, (revised version of Imperial College Technical Report DoC 90/20, October 1990).

[Abr94a] S. Abramsky, Proofs as Processes, *Theoretical Computer Science,* **135** (1994) 5-9.

[Abr94b] S. Abramsky, Interaction Categories and CSP, Draft, 1994.

[Ale94] V. Alexiev, Applications of Linear Logic to Computation: An Overview, *Bulletin of the IGPL.* Vol. 2, No. 1, pp. 77-107.

[AD96] R. M. Amadio and M. Dam, Toward a modal theory of types for the π-calculus, SICS research report R96:03.

[AGN94] S. Abramsky, S. Gay and R. Nagarajan, Interaction Categories and the Foundations of Typed Concurrent Programming, Draft, 1994.

[Avr94] A. Avron, Some Properties of Linear Logic Proved by Sematic Methods, Journal of Logic and Computation, Vol. 4, No. 6, pp. 929-938, 1994.

[BB94] J.C.M. Baeten and J.A. Bergstra, On Sequential Composition, Action Prefixes and Process Prefix, Formal Aspects of Computing 1994, 6:250-268.

[Bel90] N. Belnap, Linear Logic Displayed. *Notre Dame Journal of Formal Logic*, Vol. 31, Number 1, Winter 1990, pp. 14-25.

[Blo94] B. Bloom, When is Partial Trace Equivalence Adequate?, Formal Aspects of Computing 1994, 6:317-338.

[BdP96] T. Braüner, V. de Paiva, Cut-Elimination for Full Intuitionistic Linear Logic, Draft, April 1996.

[BS94] G. Bellin, and P.J. Scott, On the π-calculus and Linear Logic, Selected papers of the meeting on Mathematical Foundations of Programming Semantics (MFPS 92), Part 1 (Oxford 1992), *Theoretical Computer Science,* **135** (1994) 11-65.

[BW90] J. C. M. Baeten and W. P. Weijland, Process Algebra, Cambridge Tracts in TCS 18, Cambridge University Press 1990.

[Dam96] M. Dam, Model Checking Mobile Processes (Full Version), Draft, 1996.

[deO95] A. G. de Oliveira, *Proof Transformations for Labelled Natural Deduction via Term Rewriting*, (In Portuguese). Master's thesis, Depto. de Informática, Universidade Federal de Pernambuco, C.P. 7851, Recife, PE 50732-970, Brasil, April 1995.

[dG92] R. de Queiroz and D. M. Gabbay, An introduction to labelled natural deduction, In *3rd Adv. Sum. Sch. in AI*. Available from `theory.doc.ic.ac.uk`, file named `intro-lnd.{dvi,ps}.gz` in the directory `/guests/deQueiroz`.

[dG94] R. de Queiroz and D. M. Gabbay, Equality in Labelled Deductive Systems and the functional interpretation of propositional equality, *Proceedings of the Ninth Amsterdam Colloquium*.

[Fre79] G. Frege, *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen demkens.* Verlag von Louis Nebert, Halle, 1879. English translation 'Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought' in [van67], pages 1-82.

[Fre93] G. Frege, *Grundgesetze der Arithmetik. Begriffsschritlich abgeleit. I.* Verlag von Hermann Pohle, Jena, 1893.

[Gal92] J. Gallier, Constructive Logics, Part I: A Tutorial on Proof Systems and Typed λ-calculi, Draft, 1992.

[Gal95] J. Gallier, Constructive Logics, Part II: Linear Logic and Proof Nets, Draft, 1995.

[Gay94] S. Gay, A Sort Inference Algorithm for the Polyadic π-calculus, Draft.

[GN95] S. Gay and R. Nagarajan, A Typed Calculus of Synchronous Processes. Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science, 1995.

[Gab90] D. Gabbay, Editorial of Volume 1 of *Journal of Logic and Computation*.

[Gab95] D. M. Gabbay, *Labelled Deductive Systems, Part I,* Oxford University Press. (Forthcoming)

[Gd92] D. M. Gabbay and R. de Queiroz, Extending the Curry-Howard interpretation to linear, relevant and other resource logics, *The Journal of Simbolic Logic*, 57(4):1319–1365, 1992.

[Gen35] G. Gentzen, Untersuchungen über das logische Schliessen, *Mathematische Zeitschrift* 39:176-210 and 405-431. English translation 'Investigations into Logical Deduction' in *The Collected Papers of Gehrard Gentzen.*, edited by M. E. Szabo, North-Holland, Amsterdan, 1969, pp. 68-131.

[Gir87] J. -Y. Girard, Proof Theory and Logical Complexity - Volume I, Bibliopolis, 1987.

[Gir87] J. -Y. Girard, Linear Logic, *Theoretical Computer Science,* **50** (1987) 1-102.

[Gir88] J.-Y. Girard, Y.Lafont, and P.Taylor, *Proof and Types,* Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

[Gir89] J.-Y. Girard, Towards a Geometry of Interaction, in: *Categories in Computer Science and Logic,* ed. by J.W. Gray and A. Scedrov, *Contemporary Mathematics,* **92,** AMS, pp. 69-108.

[Gir95a] J.-Y. Girard, Proof-nets: the parallel syntax for proof theory. In *Logic and Algebra*, New York, 1995. Marcel Dekker.

[Gir95b] J.-Y. Girard, Linear Logic: Its Syntax and Semantics, Draft.

[Göd80] K. Gödel, On a Hitherto Unexploited Extension of the Finitary Standpoint, Journal of Philosophical Logic 9 (1980) 133-142.

[GSS92] J.-Y. Girard, A. Scedrov, and P.J. Scott. Bounded Linear Logic: a modular approach to polynomial time computability. *Theoretical Computer Science,* **97** (1992) 1-66.

[Gup94] V. Gupta. Chu Spaces: A Model of Concurrency. PhD Thesis, Department of Computer Science, Stanford University, August 1994.

[Hen88] M. Hennessy, Algebraic Theory of Processes, MIT Press, Cambridge, Massachussetts.

[HM85] M. Hennessy and R.Milner, Algebraic Laws for Nondeterminism and Concurrency, Journal of ACM, Vol.32, pp.137-161.

[Hoa85] Hoare, C.A.R., Communicating Sequential Processes, Prentice Hall, Englewood Cliffs, NJ.

[How80] W. Howard, The formulae-as-types notion of construction, privately circulated notes, only later published in *To H. B. Curry: Essays on*

combinatory logic, lambda calculus and formalism J. P. Seldin and J. R. Hindley, editors, Academic Press, London, 1980, pp. 479-490.

[Kan92] M. Kanovich, Horn programming in linear logic is NP-complete. In *Proc. 7th Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California,* pp. 200-210. IEEE Computer Society Press, Los Alamitos, California, June 1992.

[Laf88] Y. Lafont, LOGIQUES, CATEGORIES & MACHINES: Implantation de Langages de Programmation guidée par la Logique Catégorique, PhD Thesis. Universite Paris VII.

[Laf94] Y. Lafont, From Proof-Nets to Interaction Nets, Draft, June 13, 1994.

[Laf95] Y. Lafont, Undecidability of second order linear logic without exponentials, Draft, 1995.

[Lin92a] P. Lincoln *et alli.,* Decision Problems for Propositional Linear Logic, *Annals of Pure and Applied Logic*, 56:239-311, Special Volume dedicated to the memory of John Myhill.

[Lin92b] P. Lincoln, Computational Aspects of Linear Logic, PhD thesis, Stanford University, 1992.

[LW94] P. Lincoln and T. Winkler, Constant-only multiplicative linear logic is NP-complete, Selected papers of the meeting on Mathematical Foundations of Programming Semantics (MFPS 92), Part 1 (Oxford 1992), *Theoretical Computer Science,* **135** (1994) 155-169.

[Mar96] J.-Y. Marion, Additive Linear Logic with Analytic Cut. Third Workshop on Logic, Language, Information and Computation (WoLLIC'96), pp. 48-51, Salvador, BA, Brazil, May 8-10, 1996.

[Mar82] P. Martin-Löf, Constructive Mathematics and Computer Programming. In Logic, Methodology and Philosophy of Science VI. Edited by L.J. Chohen, J. Los, H. Pfeiffer & K.-P. Padewski, North-Holland, 1982.

[Mar84] P. Martin-Löf, Intuitionistic Type Theory (notes by Giovanni Sambin of a series of lectures given in Padova, June 1980), Studies of Proof Theory, Bibliopolis, Naples.

[Mie92] D. Miller, The π-calculus as a theory in linear logic: Preliminary results. Proceedings of the 1992 Workshop on Extensions to Logic Programming, edited by E. Lamma and P. Mello, Lecture Notes in Computer Science, Springer-Verlag.

[Mil80] R. Milner. A Calculus of Communicating Systems. *Lecture Notes in Computer Science 92,* Springer-Verlag, 94 pp.

[Mil83] R. Milner, Calculi for Synchrony and Asynchrony, *Theoretical Computer Science*, **25** (1983) 267-310.

[Mil84] R. Milner, Using Algebra for Concurrency. In *Distributed Systems,* ed. by Chambers & Duce, Academic Press, 1984, pp. 291-305.

[Mil86] R. Milner, A finite delay operator in Synchronous CCS, Internal Report CSR-116-86, May 1986, University of Edinburgh.

[Mil93] R. Milner. The polyadic π-Calculus: a tutorial, in: *Logic and Algebra of Specification,* eds. F.L. Bauer, W. Brauer, and H. Schwichtenber

[San96] D. Sangiorgi, An interpretation of Typed Objects into Typed π-calculus, Rapport de Recherche, Institut National de Recherche en Informatique et en Automatique, September 1996.

[Sch94] H. Schellinx. The noble art of Linear Decorating. Amsterdam: Universiteit van Amsterdam, Faculteit Wiskunde en Informatica. - (ILLC dissertation series; no. 1).

[WN95a] G. Winskel and M. Nielsen, Models of Concurrency. In vol. 4 of the Handbook of Logic in Computer Science, Oxford University Press, 1995. A draft appear as BRICS Report RS-94-12.

[WN95b] G. Winskel. and M. Nielsen, Models of Concurrency. Summer School on Semantics and Logics of Computation. 25-29 September 1995.

[Tan92] A.S. Tanembaum, Modern Operating Systems, Prentice-Hall International Inc., 1992.

[Tro??] A.S. Troelstra, *Tutorial on Linear Logic.* In: *Substructural Logics.* Series Editor: Dov M. Gabbay. Editors: Kosta Došen, Peter Schroeder-Heister.

[van67] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic. 1879-1931.* Series *Source Books in the History of Sciences.* Harvard University Press, Cambridge, Massachussetts, xii+664pp, 1967.