Geometry of Parallelism: A Uniform Analysis of Evaluation Strategies and Effects by Synchronous Interaction Abstract Machine

(並行性の幾何: 同期付き相互作用抽象機械による評価戦略および副作用の統一的な解析)

by

Akira Yoshimizu
由水輝

A Doctor Thesis
博士論文

Submitted to
the Graduate School of the University of Tokyo
on December 9, 2016
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Thesis Supervisor: Ichiro Hasuo　蓮尾一郎
Associate Professor of Computer Science

ABSTRACT

This thesis introduces the notion of *Synchronous Interaction Abstract Machine* that is a novel variation of the Geometry of Interaction (GoI) framework. We provide a thorough account on the newly introduced framework and employ it to obtain adequate semantics of a class of programming languages possibly with memories and effects, uniformly treating the call-by-name and the call-by-value strategies.

The *Geometry of Interaction* (Girard, LC 1988) is originally a scheme to give dynamic semantics to linear logic proofs. Later, it has proved to be a useful framework in computer science in general: when represented as a token machine called the *Interaction Abstract Machine (IAM)* (Danos & Regnier, Electr. Notes Theor. Comput. Sci. 1996), it can be seen as a compilation scheme of possibly higher-order computation into first-order, low-level computation. This character of GoI is theoretically interesting and at the same time provides a compiler implementation technique based on semantics (e.g. Mackie, POPL 1995). As such, GoI has resulted in a number of results in computer science such as programming language semantics (Hoshino, FoSSaCS 2011), optimal reduction strategy (Gonthier et al., POPL 1992), and defunctionalization (Schöpp, TLCA 2013).

The main achievement of the thesis is to better understand the capability of GoI (and in general dialogue-based) semantics compared to other kinds of semantics. It is presented via interpretation of a quantum programming language with higher-order functions and recursion by an extension of the GoI token machine semantics, namely utilizing *multiple* tokens at a time. Establishing such a *multi-token machine* framework itself is non-trivial and is one of the main contributions of the thesis. Moreover, a seemingly different result, namely distinction of the call-by-name and the call-by-value behaviors turns out to follow from the same notion of multi-token machine. We expect that the concurrent and fine structure of our multi-token machine also provides possibilities of applications to semantic approach to inherent parallelism of functional programs and a unified framework for type systems for concurrent computation. Concretely, the contributions of this thesis are as follows.

In the thesis, we provide a thorough account on a multi-token machine on a much more expressive proof net system (that we call *SMEYLL proof nets*) that accommodates higher-order functions, recursions, and branchings. For linear calculi such a multi-token machine have been recently proposed (Dal Lago & Faggian, QPL 2011, Yoshimizu et. al., ESOP 2014), but compared to the linear cases it turns out to be much more non-trivial whether such a multi-token system behaves well, e.g. whether the machine can deadlock or not. We examine properties on the proof net system, the multi-token system, and the relation between the two, by using both traditional notions in linear logic and new proof techniques we introduce. As a result, it is shown that those systems behave surprisingly well despite its inherent complexity. The main results include deadlock-freedom of the SIAM, invariance and adequacy of the multi-token semantics with respect to net reduction.

Using those results shown, we interpret a PCF-like calculus by SMEYLL nets and the SIAM (appropriately extended with a notion of integer memory). The SIAM is shown to possess superiority over the standard IAM: our SIAM can distinguish the call-by-name and call-by-value reduction strategies without any special construction other than the two corresponding translation of calculus into proof nets, while the IAM (or GoI model in general) can only exhibit the call-by-name character even if we naively apply the call-by-value translation. This is made precise as an adequacy result between the calculus and the extended SMEYLL nets, which is in turn adequately interpreted by the extended SIAM. The fact that such a parallel, non-standard token machine can adequately interpret a standard sequential calculus is already of theoretical interest, and moreover the parallelism is the source of enrichment of the distinguishing power on reduction strategies.

Furthermore, we again extend the SMEYLL proof net system and the SIAM system by equipping them with a notion of memory structure and (possibly) probabilistic transitions depending on contents of a memory. Instances of memory structures include deterministic, probabilistic, and quantum memories. The systems are again shown to satisfy desirable properties in the probabilistic setting; to prove those properties, we also introduce a probabilistic variation of abstract reduction system, of which the definition is

of independent interest. Finally we interpret a calculus equipped with a memory structure by SMEYLL proof nets and the SIAM with a memory structure. All the proofs are done in a parametric way with respect to memory structure, thus we obtain adequacy result for all the three instances (and any other instance) at a time.

論文要旨

本論文では Geometry of Interaction (GoI) の新たな変種である *Synchronous Interaction Abstract Machine* を導入し, 詳細な分析を与えた上でそれを用いてメモリおよび副作用の概念を持ちうるあるクラスに属するプログラミング言語の名前呼び・値呼び戦略の双方に対して妥当な意味論を統一的に与える.

Geometry of Interaction (Girard, LC 1988) は元来線型論理の証明に対する動的な意味論を与えることを目的とした枠組みであるが, 後にプログラミング言語理論一般においても有用な枠組みであることが分かってきた. GoI は *Interaction Abstract Machine (IAM)* (Danos & Regnier, Electr. Notes Theor. Comput. Sci. 1996) と呼ばれるトークン機械としての表現を通じて, 高階計算を 1 階の低レベルな計算へコンパイルする枠組みだと見ることができる. この特徴は理論的に興味深いと同時に 意味論に基づいたコンパイラの実装技法を与え (Mackie, POPL 1995 など), その他プログラミング言語意味論 (Hoshino, FoSSaCS 2011), 最適簡約戦略 (Gonthier et al., POPL 1992), defunctionalization (Schöpp, TLCA 2013) などに応用がある.

本論文の主たる貢献は, 高階関数・再帰を持つ量子プログラミング言語の解釈という結果を通じ, GoI 意味論 (より一般に対話ベースの意味論) の特性についてのより良い理解の提供したことである. 本論文はこの課題を, トークン機械意味論において複数のトークンを用いるという拡張によって解決する. そのような複数トークン機械の意味論の設計・確立それ自体も非自明なものであり, 本論文の主要な貢献である. さらに, この拡張が名前呼び・値呼び戦略の識別という一見して異なる問題についても統一的に解決を与えることを示す. また複数トークン機械の並列性と詳細さは関数型プログラムの inherent parallelism に対する意味論的なアプローチ, 並行計算の型システムに対する統一的なフレームワークの提供などに応用できることが期待される. 具体的な貢献は以下のようになる.

本論文では, 高階関数・再帰・分岐を含むより表現力の高い proof net (*SMEYLL proof net*) 簡約系に対する複数トークン機械 *Synchronous Interaction Abstract Machine (SIAM)* について詳細な分析を与える. 上記の線型な場合に比べ, デッドロックするか否か等のトークン機械の振る舞いの解析は遥かに非自明なものとなる. 本論文は proof net 簡約系とその上での複数トークン機械の性質, およびこれら 2 つの関係について, 線型論理における既存の概念と新たな証明技法の双方を用いて解析を行い, 結果として内在する複雑さに反してこれらが良い振る舞いを示すことを示す. 主結果としては複数トークン機械のデッドロック自由性, 複数トークン機械の proof net 簡約についての不変性および妥当性が挙げられる.

これらの結果を用いて, PCF に近い言語の解釈を整数を保持するメモリを適切に付与した SMEYLL net および SIAM によって行う. これにより, 言語の解釈において, SIAM は名前呼び・値呼び戦略を proof net への翻訳のみによって行えるという利点があることが示される. これは標準的な IAM が (少なくともナイーブには) 名前呼び戦略のみしか扱えないことと対照的であり, 並行性を持つ非標準的なトークン機械である SIAM が逐次的な言語の解釈においてこのような意味を持つことは理論的に興味深いものである.

さらに SMEYLL proof net および SIAM に対し, メモリ構造と名付けた概念およびメモリの内容に依存した確率的な遷移を付与する. 抽象概念であるメモリ構造はインスタンスとして通常の決定的なメモリ, 確率的メモリ, 量子メモリを内包する. このメモリ構造を

持った SMEYLL proof net, SIAM はやはり妥当性・不変性などの良い性質を持つことが確率的な設定のもとで示される. 証明には抽象書換え系のやはり確率的な拡張である確率的抽象書換え系の概念を導入して用いているが, この確率的抽象書換え系の概念自体も理論的には興味深いものである. 最後に, メモリ構造を含んだプログラミング言語をメモリ構造を含んだ SMEYLL proof net および SIAM により解釈し, その妥当性を示す. この解釈および上記の性質の証明は全てメモリ構造をパラメータとして扱っており, 各インスタンスについての性質は全て同一の定理から従う.

Acknowledgements

Thanks are first of all due to Claudia Faggian, Ugo Dal Lago and Benoît Valiron, not only for carrying out joint research (that led to several papers) with me but also warm and friendly communication in Paris and Bologna when I visited them. I would like to thank all the current and former members of Hasuo laboratory and ERATO Hasuo MMSD project. I am gratefully indebted to my parents Shin and Michiko for kindly watching me and always welcoming me when at home. I sincerely thank the examiners of this thesis for allowing me an opportunity to reflect on the arguments in the thesis. Finally, I would like to express the deepest gratitude to our supervisor Ichiro Hasuo. I could not arrive here without his patient and continuous encouragement.

This work has been partially supported by Grant-in-Aid for JSPS Fellows, the JSPS-INRIA Bilateral Joint Research Project CRECOGI, and JST ERATO Grant Number JPMJER1603.

Contents

Chapter 1

Introduction

1.1 Background

Programming language semantics. Understanding computation is one of the principal aims of computer science. By representing individual computation as a program, *programming language semantics* describe the behavior of the program in a formal, mathematical language. A programming language semantics provides us a rigorous description of "what does this program do?" and therefore allows us to examine whether a program behaves as intended or not; such an examination based on mathematical description is called *formal verification*. A formally defined programming language semantics can also be viewed as a guideline for compiler implementation of the language, or sometimes a semantics even directly derives techniques for implementation. Thus studying programming language semantics has both theoretical and practical benefits. The practical benefit of formal verification becomes significant when the program in consideration is inherently hard to analyze by *testing*, i.e. by feeding sufficiently many inputs to the program and checking whether the outputs coincide with our expectation. In a *probabilistic programming language*, there can be a rare but critical event which may be overlooked with high probability by testing. In a *quantum programming language* the situation becomes even worse because testing itself costs large amount of resource due to the current hardware limitation. For such languages with *effects*, the right way should be formal verification based on rigorous language semantics. Indeed, seeking semantics for those languages with various effects has been one active research area [41, 66, 68, 74].

There has been much work on programming language semantics since the early days of computer science, and semantics proposed so far are classified into three groups: operational, denotational, and axiomatic semantics. An *operational semantics* defines how to evaluate a program step by step, either as a direct transition relation over programs or as an abstract machine. This view of computation as state transitions is suitable for implementation and agrees with programmers' intuition. λ-calculus with the standard β-reduction rule and the SECD machine [64] are typical examples of this kind. A *denotational semantics* assigns a mathematical object (e.g. a number, a set, or a function; called *denotation*) in a certain semantic domain to each program. This assignment is usually done in a *compositional* manner in the sense that the denotation of a composite program is obtained from the denotations of its subprograms. An advantage of denotational semantics is easiness of reasoning thanks to this compositionality. Domain theory [89] is a fundamental denotational semantics where a function (in a program) is interpreted as a continuous function (in the mathematical sense) over a domain. Another kind of programming language semantics is *axiomatic*

semantics that describes the behavior of a program as predicates that hold before and after the execution of the program. A representative of such semantics is Hoare logic [48]. Given a programming language and its axiomatic semantics, finding a derivation of specifications written as predicates is nothing but program verification.

Among them, there is a family of semantics based on *interaction dialogues*, including *game semantics* [6, 53] and semantics via *Geometry of Interaction*(*GoI* for short) [39]. They can be seen as denotational semantics that at the same time carry operational nature. On the one hand, a dialogue-based semantics is denotational in the sense that the semantics represents a program as a mathematical object, that is a strategy of a player in a game (in game semantics) or flows of information over the program (in GoI). On the other hand, it is operational in the sense that denotations are calculated by (or can be seen as) a dynamic procedure of strategy composition (in game semantics) or path computation (in GoI). The combination of these characters lead to some distinctive features of dialogue-based semantics: full abstraction results by game semantics [6, 53], fine control on resource consumption [36, 63], and compiler implementation based on denotational semantics [35, 69].

In particular we focus on GoI semantics in this thesis. To explain our aim, we first elaborate the intuition of GoI. Geometry of Interaction is a semantics of linear logic proofs that was originally invented by Girard [39] to characterize not only the proofs themselves, but also the *dynamics* of the procedure of *cut-elimination* on the proofs. Intuitively, in GoI semantics, a proof is interpreted by a set of certain "paths" on the proofs. After one step of cut-elimination is performed, the shapes of paths change accordingly (hence dynamics of cut-elimination is captured), but at the same time the correspondence of formulas in the conclusions of the proof remains the same (hence the paths form a semantics). From the perspective of computation, it can be rephrased as: the meaning of a program is interpreted as a certain *information flow* in the program, and when the program is reduced to an equivalent one the flow itself (of course) changes, but the correspondence of inputs and outputs remains the same. The dynamics of GoI stands out when represented as a path computation by so-called *token machine* [23] or *context semantics* [42]; especially we rely on the formulation of token machine in the thesis. In a token machine, a run of the machine finds out a path satisfying the above-mentioned character. GoI also has a characteristic that even a higher-order computation is reduced to first-order one. It is intuitively clear in token machine semantics since a token can only carry basic, local information on some stacks, without any pointers, variables to be substituted, or thunks. It is indeed shown that a semantics closely related to GoI is a composition of *defunctionalization* and continuation passing style transformation [87]. Later it is proved that GoI yields the notion of linear combinatory algebra via its categorical formulation [4], and it is used to give semantics for programming languages [50, 69].

1.2 Aims of the thesis

The main aim of this thesis is *a better understanding of characters of dialogue-based semantics*. To this end, the thesis introduces an extension of the GoI framework that computes *all* the paths relevant to a proof net by exploiting *multiple* tokens rather than one single token *at a time*. In this *multi-token machine semantics*, possibly many tokens move around interacting with each other; thus such a multi-token machine inevitably gains parallel, concurrent nature (it is however in a very organized way, as we will see in the thesis). Indeed, a pos-

sible gain of parallelization of GoI was recently mentioned in a paper by Hasuo and Hoshino [45]. They give a GoI-based semantics for an expressive quantum calculus in the paper; while it is remarkable work that provides one of the first adequate denotational semantics for such a higher-order quantum calculus, the calculus has a syntactic restriction on the handling of entangled quantum bits. The fact that only a single token is exploited in their work is mentioned as the source of that restriction. Interestingly, a closely related, dialogue-based approach, namely quantum game semantics proposed by Delbecque [24] also holds a similar constraint. Such a phenomenon is not present in other work on quantum programming language semantics, for example in the one by Pagani et al. [79]. In general, dialogue-based semantics are known to be good at yielding fine-grained models, but for some reason it is not true in this particular application. A natural question here is: is it an essential limitation, or is there room for ameliorating the models concerning this point? We figure out that the essential reason why such a restriction arises in Hasuo and Hoshino's work is: without the restriction, there may be some paths that are relevant to a correct interpretation but are missed by their token machine (and any of existing standard single-token machines). This thesis approaches the question based on this insight. By answering this question, we better understand pros and cons of dialogue semantics and other kinds of semantics.

It is notable that there exists another problem with the same root. The *call-by-name translation* and *call-by-value translation* are two well-known encodings of λ calculus with corresponding evaluation strategies into multiplicative exponential proof nets [38]. It is also known that, although both encodings are sound w.r.t. reduction of λ terms, the standard GoI semantics of those proof nets only characterizes the call-by-name behavior. In other words, the GoI semantics of the call-by-value translation is not sound if naively applied. In fact, this phenomenon can be explained in the same way as the above-mentioned quantum case: the gap between proof net reduction and the GoI semantics arises from the missing paths in the computation. Therefore, remedying this gap solves two seemingly different problems uniformly.

Moreover, we envisage that the idea of multi-token machine semantics can exhibit its advantages in at least two more areas of computer science: *inherent parallelism* and *concurrent computation*. These topics are beyond the scope of this thesis (though the research is ongoing and a partial result has been obtained on the latter; see Chapter 6). The motivations and results we expect are summarized in the next two paragraphs:

Implicit parallelism from a semantic viewpoint. Functional programming languages are said to be suitable for writing parallel programs [59], and there has been research on *implicit parallelism* of functional programs. Since functional programs have referential transparency, it has to be safe to execute different subparts of the program, and thus execution of functional programs can be accelerated by that kind of parallelization. Despite the clarity of the idea, it still seems to remain as a folklore and not much success has been seen in the automatic generation of such parallel compilation [96], partly due to non-trivial behaviors of higher-order functions [49]. A multi-token machine applied to functional languages may shed light on the subject from a semantic viewpoint, by its nature to decompose higher-order computation into first-order token transitions together with the newly introduced parallel character.

Uniform account for type systems in concurrent computation. A naturally expected application from such a multi-token machine formulation is an interpretation of *concurrent computation*. Concurrent processes and interactions between

processes are often modeled as terms in *process calculi*, for example CCS [71] or π-calculus [73]. Then properties desired about such processes, e.g. deadlock-freedom, can be assured by type systems for those process calculi [62, 98]. By interpreting process calculi with multi-token GoI machines, the syntax-free nature and the origin being semantics for logic may allow us to uniformly account for various type systems for those calculi. This will lead to better understanding of the vast amount of existing calculi and type systems as well as new derivation of type systems based on a unified semantic framework.

1.3 Contributions

Despite the simplicity of the idea, modeling (sequential) programming languages with such a parallel multi-token machine is far from straightforward. In this situation the most natural way to dispatch data to those multiple tokens is to make one token carry one data. When an n-ary function is applied on those data, the corresponding tokens are *synchronized* and data are modified accordingly. In general, the setting with multiple tokens synchronizing with each other immediately leads to a possibility of *deadlocks*. Moreover, when the language we would like to interpret accommodates duplicable data, we have to be able to control the number of tokens that correspond to usage of each data: a (structure corresponding to) data used twice must generate two distinct tokens, while a data that is discarded must not generate any token. Such a feature is not present in the standard token machine. Further, the situation with many tokens running simultaneously, synchronizing with each other, is much like that of *Petri nets* [82]. It is known that the behaviors of Petri nets are in general hard to verify or analyze: for example, decision problem of deadlock-freedom of Petri nets is known to be in EXPSPACE, and in PSPACE-complete for even 1-safe Petri nets that is a rather strong restriction (for more details on those decision problems on Petri nets, see [30, 31]).

Thus whether such a multi-token system with synchronization can be a model of computation or not is, even without computational effects, a non-trivial problem. Being (perhaps surprisingly) new in the research area of Geometry of Interaction, the following questions arise:

- Can we rigorously reason about such a system?
- Can such a parallel system be a computational model of sequential computation in any sense?
- If it can be so, can it achieve the above-mentioned aims?
- Is there any other interesting notion that is covered by a multi-token system but not by a single-token one?

The thesis will deal with this non-standard notion of multi-token machine, and give a thorough account on the definition, properties, and resulting models of computation. A summary of the contributions of this thesis is as follows. The first three are described in Chapter 4, and the last two are in Chapter 5.

- *Introduction of a notion of multi-token machine.* First of all we define a precise definition of a *multi-token machine*, called the *Synchronous Interaction Abstract Machine* (SIAM for short), together with an extended notion of proof nets, called *SMEYLL proof nets*. The SMEYLL proof nets include three novel kinds of nodes to handle *synchronization, branching*, and

recursion respectively. The definition that allows a rigorous modeling of programming languages in the setting is already novel and not straightforward to find out. A distinctive feature of the SIAM is that the machine generates tokens from unit nodes and dereliction nodes on the fly, and such generation is itself controlled by the tokens generated from dereliction nodes. The machine gives a *token machine semantics* that is a partial function computed by the machine to the nets.

- *Investigation of properties of the multi-token machine.* We prove various properties about the nets and the machine we define. The properties of the machine are either far more non-trivial to prove than the case of the standard Interaction Abstract Machine, or are simply not present in the standard case. Among them, *deadlock-freedom* is a distinctive and important one: by definition it is absent in the standard single-token machine, and without a guarantee of being free from deadlocks we are unable to assure that the token machine semantics makes sense or not. The pleasant properties can be established in the complicated, multi-agent system of the SIAM by taking advantage of being based on linear logic and proof net techniques. Relying on polarity of logical formulas is especially useful here.

- *Uniform account of the call-by-name and the call-by-value strategies.* The nets and the machine are applied to interpret a PCF-like calculus, and an adequacy result of the interpretation is shown. Such a semantics of a sequential language by a parallel, concurrent system is itself of theoretical interest. Moreover, it turns out that our multi-token machine can not only form an adequate semantics of the calculus, but also provides a solution for the problem inherent to the standard GoI, namely distinction of the *call-by-name translation* and the *call-by-value translation* of calculi into linear logic proof nets. That is, our machine can *uniformly* interpret both the call-by-name and the call-by-value translations of terms into proof nets thanks to the characteristic of multi-token machine that computes *all* the paths in parallel, while those paths computed by the standard one only matches the behavior of the call-by-name case (at least naively).

- *Parameterization by memory structures.* Furthermore, the initial purpose of introducing multiple tokens is accomplished with a satisfactory expressiveness and a certain generality: we interpret another PCF-like calculus parameterized by *probabilistic branching effects*. Those effects and the content of memory along execution are axiomatized as a notion called *memory structure*. The adequacy result of interpretation is also proved *parametrically* on memory structures. As instantiation of the memory structure, we automatically obtain semantics of deterministic, probabilistic, and quantum calculi. In the quantum instance, the syntactic constraint found in [24, 45] are not present and qubits can be separately treated regardless of entanglement. The deterministic case essentially coincides with the usual PCF case, and the probabilistic case is akin to existing probabilistic calculi, e.g. [26].

- *General properties of probabilistic abstract reduction system.* As a byproduct, we introduce a probabilistic extension of abstract rewriting systems that is also novel and of independent interest. Probabilistic extension is a natural direction both for syntax and for semantics of programming languages, as done for programming language [86] and for semantics e.g. domain theory [57] or game semantics [21]. However, such a probabilistic

extension seems to be undeveloped so far for abstract reduction systems that is also a common and general notion in the programming language community. The notion of parallel rewriting of elements in a distributions is novel, and a property called *uniqueness of normal forms* can be stated on the newly introduced rewriting relation over distributions. The property does not have its counterpart in the usual abstract rewriting systems; it makes sense only in the probabilistic setting, and we rely on that property in a proof about our probabilistic nets and machine.

Organization of the Thesis. Related work to this thesis is discussed in Chapter 2. Chapter 3 provides preliminaries on the basic mathematical or logical notions on which we build our work. Chapter 4 and Chapter 5 are two main chapters of the thesis. Chapter 4 introduces the notions of SMEYLL proof nets and the Synchronous Interaction Abstract Machine (SIAM) executed on those nets. The progress property and the cut-elimination property of SMEYLL proof nets, as well as deadlock-freedom, invariance, and adequacy of the semantics given by SIAM are proved with novel proof techniques. Then we apply the framework to a basic but expressive language akin to PCF by translating terms into proof nets, and its semantics is obtained by executing the (extension of) SIAM on the nets translated from terms. The translation is shown to be adequate, and hence via the translation an adequate multi-token GoI semantics of the PCF-like language is accomplished. Chapter 5 further extends the proof net system, the multi-token machine, and the PCF-like language by equipping them with a notion called memory structure. This yields a class of proof net systems, multi-token machines, and languages; the adequacy results are also proved in a parameterized way. The results are shown to subsume the case shown in Chapter 4 as well as a quantum version of the three systems as a notable instance. Chapter 6 concludes the thesis, and lists some possible directions of future work.

Chapter 2

Related Work

In this chapter, we review some papers relevant to our work in the thesis. Section 2.1 describes related work in terms of the specific approach we take in the thesis, namely Geometry of Interaction and its token machine representation. Section 2.2 contains some other approaches toward the (short-term and long-term) aims we stated in Chapter 1.

2.1 Work Related to Our Approach

2.1.1 Game Semantics

Game semantics was originally devised for logics [67] and later applied to a programming language called PCF [6, 53]. One important capability of game semantics as language semantics is that it yields *fully abstract* semantics: equivalence in the model not only implies observational equivalence of the programs corresponding programs, but also the converse holds (i.e. the equivalence in the game model and the observational equivalence of programs coincide). Full abstraction is a considerably harder property to obtain than soundness or adequacy, and construction of such a model for PCF was a long-standing open problem.

Game semantics and Geometry of Interaction are frameworks to give denotational semantics based on dialogues: a game semantics explicitly employs two players' dialogues as programs' behaviors, and GoI semantics can also be formulated as input/output dialogues between components via Int construction [4]. A distinctive feature of these frameworks is that they are *dynamic* at the same time providing denotational (*static*) semantics. Since the use of computation and information is getting more and more dynamic today, this perspective has certain importance [2]. Describing such a dynamic process as a mere mathematical function is here inappropriate, and dialogue-based semantics would serve as a foundation for such a theory of dynamic information.

In a basic case of modeling multiplicative linear logic (MLL) the two semantics indeed coincide [5]. On the one hand, game semantics has larger design space in its definition. Relaxation or extension of the definition of games and strategies are very often done in order to model various features and effects in programming languages, and those semantics with such fine tuning often lead to full abstraction results. On the other hand, GoI has more direct connection to linear logic. Those semantics called Geometry of Interaction should be a model of linear logic or its extension, so the design space is relatively restricted when compared to game semantics. Instead, an advantage of GoI is its simplicity: computation is after all a run of a token with some elementary data, following some simple rules.

As mentioned in Chapter 1, a programming language semantics is often useful for compiler implementation, either as a guideline or as a practical derivation

method. Geometry of Interaction is known to provide us the latter aspect [69], and a variant of game semantics is also suitable for mechanization [13]. Although other denotational semantics can also derive such [58, 100], the dynamic and fine-grained nature of GoI/games often yields a notable application. For example, IntML [63] language made from an observation on dialogue semantics has a distinctive feature that functions with sublinear space can be expressed. Another is a compiler called Verity derived from a series of paper entitled Geometry of Synthesis [35]: it compiles high-level functional languages even with recursion into hardware description, with modest efficiency and a background on a variant of game semantics. Speaking of our work, it is currently not clear whether it leads to some concrete compilation technique or not; however, combining the concurrent utilization of multiple tokens in our work with the work by Fredriksson and Ghica's compiler for distributed computation based on GoI [32] seems natural and possibly show some advantage over compilers developed in a usual way (e.g. automatic and efficient parallelization based on our semantics could be one possibility).

2.1.2 Petri Nets

Though the motivation on concurrent computation in this work arises from observation on process calculi, the work in this thesis is definitely reminiscent of another formalism, namely Petri nets (especially of *colored Petri nets* [56], since each token carries some data). The setting that multiple tokens run around a directed graph structure, sometimes synchronizing with each other, is common to Petri nets and our approach. However, there is a difficulty to treat the multi-token machine we will define in the thesis as Petri nets. That is, if we aim at formalizing our token machine as a colored Petri net, it is necessary to take an infinite set of stacks as a color set and also to employ infinitely many tokens. Such an infinite representation is usually prohibited in Petri net theory, and thus it raises another layer of difficulty to analyze the machine. It is also in contrast to the fact that the number of tokens in our machine is always finite (even though the number is unbounded).

There also exist some papers that give semantics of linear logic (on which we strongly rely) by means of Petri nets [29, 34]. Those semantics clearly explain the "logic of resource" view of linear logic via Petri nets. The difference from our token machine semantics (or from GoI semantics in general) is that their semantics are proof-irrelevant ones, so the dynamics within those Petri nets is not the same as what is achieved by the token machine approach taken in the thesis. Concerning our contribution in this thesis, how to extend their semantics to appropriately reflect evaluation strategies is also unclear.

2.1.3 Concurrent Computation and Geometry of Interaction

One of the first attempts to introduce multiple tokens in the framework of token machine is the work by Dal Lago and Zorzi [17]. The notion of synchronization, applying an operation to a memory accompanying the state, is already present there. However, both expressiveness and generality are rather restricted in their work: their token machine cannot interpret measurement or non-linear data, and the setting is specific to quantum setting.

A paper following them [99] (that is mostly the work done as the author's master thesis) can be seen as an extension of the work [17] to accommodate measurement using a structure akin to the $\bot$ introduction rule in linear logic. While

it is true that it interprets all the basic ingredients of quantum computation, the token machine mechanism to interpret measurement is in a sense *static*: the machine first specifies all the branching it follows, then calculate the probability and the resulting state of that branch. If we consider the actual execution of such a quantum program, what is natural is to determine the branching *dynamically*, one by one. Thus, besides the fact that it is still based on the multiplicative fragment, it is unsatisfactory as a dynamic semantics of quantum programs (or more generally, programs with effects).

As for synchronization of tokens, Asperti and Dore [9] also studied the relation between linear logic proof nets and synchronization of tokens. However their work has been done from a different point of view compared to what will be done in the thesis. In their work, tokens synchronize solely to establish a path and then they just disappear; the process of synchronization deadlocks if and only if the underlying proof structure is not correct, i.e. not a proof net. (The picture would be clearer after we show some intuitions and definitions; see also Chapter 4.) Hence, while it is true that viewing a multi-token machine as a distributed system can be interesting, the token machine they use does not suit the purpose of the thesis.

The work by Fredriksson and Ghica [32] (which we already mentioned in this chapter) is also at the intersection of Geometry of Interaction and distributive systems, but in a different way. They implemented a compiler from PCF terms into distributive C codes with the MPI library. From the viewpoint of nets and GoI, it is not the *abstract* token machine itself but the *actual* machines executing computation that is distributive; the token machine used there is a sequential, single-token one based on Hoshino [50] and Mackie [69].

2.1.4 Evaluation Strategies and Geometry of Interaction

When applied naively, a token machine of Geometry of Interaction can only interpret the call-by-name evaluation strategy. Intuitively, this is because a path regarded as a meaningful one connects one output of a program with only a substructure of the underlying net/program that is relevant to the output. Typically it means that if the output is irrelevant to an input then the path computing the output does not even touch the input (as in the call-by-name strategy), while in the standard call-by-value strategy the input is necessarily evaluated. As known, the difference can cause different behaviors of programs, notably shown by the one like $(\lambda x.\overline{1})\Omega$ (where Ω is a diverging term): the "output" $\overline{1}$ of the term is irrelevant to the input Ω of the lambda abstraction $\lambda x.\overline{1}$, so it converges in the call-by-name while diverges in the call-by-value strategy.

Existing work such as one by Hoshino [50] (and consequently by Hasuo and Hoshino [45]) or one by Schöpp [88] uses variants of the continuation passing style (CPS) transformation by Plotkin [85] to impose the call-by-value evaluation order on the execution path of GoI. The CPS transformation is a well-known technique and also known to be suitable for an intermediate language during compilation [8]. Though, it appears an indirect way when we consider the origin of GoI: there exists a standard translation of simply-typed λ calculus types into linear logic types, both in the call-by-name and the call-by-value manners. The proof nets obtained by those standard translations do reflect the difference between the two evaluation strategies, while the usual token machine on those nets exhibits the same call-by-name behavior on the two different translations. A novelty in our work about this topic is that we have found a machine that is able to detect the difference of the two in a uniform way.

2.1.5 Quantum Computation and Geometry of Interaction

The notion of quantum computation was first proposed in 80's, and especially after the finding of Shor's algorithm [94] that shows an exponential speedup compared to the known best algorithm for the integer factorization problem, much interest and effort have been put into the study of quantum computation. Although a sufficiently large quantum computer is still to be developed, many quantum algorithms that perform quadratically or exponentially better than any other known classical algorithms have been found [16,44,65], as well as quantum-based cryptography method utilizing the no-cloning property has emerged [12].

Those quantum algorithms are most often described by means of *quantum circuits*. Programming with quantum circuits is much like programming with logical circuits; such a representation is harder to analyze or verify structurally, and writing a large circuit by hand is simply tedious. Hence *quantum programming languages* have been considered from around 2000 [61, 78], and recently some compilers that compile programs written in those languages into (textual representation of) quantum circuits is developed [43, 97]. Hence, to exploit advantages of such structured, high-level languages, it would be helpful to provide rigorous semantics of those languages. Those semantics have been investigated recently [91], but still the study is in its infancy rather than at a stage that a standard one is established.

Interpretation using the framework of Geometry of Interaction is one candidate for such a semantics for quantum languages and has been sought recently; an explicit mention has been done by Scott in 2004 [90], and perhaps we could even say that it has been from around when linear logic (and consequently Geometry of Interaction) was born, since Girard himself explicitly mentions the possibility to utilize linear logic to interpret quantum mechanics (and physics in general) in 1989 [40]. We already mentioned some recent studies on applying linear logic and Geometry of Interaction in Introduction. We describe below a detail on the "non-compositional" syntax of [24, 45] and also mention another line of research on the relation between quantum computation and GoI.

Given a term M of type $\mathbb{Q} \otimes \mathbb{Q}$ (where $\mathbb{Q}$ is a ground qubit type) and two unitary gates U, V, the term (or an equivalent term of) $\mathtt{let}\ \langle x, y\rangle = \mathrm{M}\ \mathtt{in}\ \langle \mathsf{U}, \mathsf{V}\rangle x \otimes y$ is not valid in the languages in the existing work [24, 45]. This is because the constructor only accepts the product type $\mathbb{Q} \otimes \mathbb{Q}$ that is for sure not entangled, although it is an arguably common construction when we consider a program as a quantum circuit.

In the thesis, we interpret a quantum calculus with higher-order functions and recursion. The calculus is fairly expressive. On the one hand, the fact that we could avoid the syntactic constraint present in existing work [45] suggests that quantum GoI can still be a candidate for a semantics of a "practical" quantum programming language. On the other hand, as for expressiveness our multi-token semantics does not show a superiority to other quantum programming language semantics.

The paper by Abramsky and Coecke [3] relates quantum processes and categorical Geometry of Interaction from a different perspective. In the paper they apply $\mathcal{G}$ construction to the traced monoidal category FDVec of finite-dimensional vector spaces in which quantum processes can be described. The resulting Geometry of Interaction model is different from the standard token machine model. Although the interpretation of linear logic by using the notions of quantum processes is theoretically of interest, it seems a roundabout way since a mere identity represented by axiom-cut redex is realized by creating an entangled pair and pro-

jecting them.

2.1.6 Effects in Geometry of Interaction

We interpret PCF-like languages with a certain class of effects by our multi-token machine. There is a series of work in the same line by Hoshino, Hasuo, and Muroya [51, 76], called memoryful Geometry of Interaction. The languages to which they give categorical semantics also include !-types and recursion. The class of effects that memoryful GoI accommodates seems strictly larger than ours: non-deterministic choice (without probability) is not allowed in our framework while memoryful GoI can naturally handle it, and the instances of our memory structure we found so far are instances of memoryful GoI, too.

An apparent difference between our work and theirs is that their token machine is single-token while ours is multi-token. Thus our machine has a possibility to exploit the parallelism brought by being multi-token, and indeed there is an advantage over them. In memoryful GoI, by using a CPS-like construction, the call-by-value strategy is interpreted in the sense that the denotational semantics is adequate. However, the cost of computation is still that of call-by-name, meaning that even after a subterm is once evaluated, the path to evaluate it is again used in the execution later. Such a re-computation is skipped in the usual notion of call-by-value, and the behavior is indeed counted as an advantage of employing call-by-value. Our machine naturally reflects the character: once a computation of a value of ground-type finishes, the value is just copied to the tokens that are required to have the same value without doing re-computation. From this perspective we claim that our work is more fine-grained one, although its practical advantage is still to be further developed.

2.2 Work Related to Our Aims

2.2.1 Languages for Quantum Computation

Quantum process calculi (e.g. [33] and [60]) are formal languages to describe quantum processes involving communication, such as quantum teleportation or quantum cryptography protocols between two distant systems. Like the classical case, quantum process calculi are designed to describe and verify multi-process, concurrent or distributed quantum computation, while the languages like quantum λ-calculus [92] are for general description of quantum algorithms. As explained in Chapter 1, the reason we try to interpret such general-purpose quantum programming languages rather than quantum process calculi is simply that the semantic problem we aim at arises from the former. This does not imply that we cannot apply our multi-token machine to quantum process calculi, but it would be after we examine the possibility of application of our approach to non-quantum process calculi (that is one of the future directions of the work), which is too far to start a concrete discussion.

2.2.2 Implicit Parallelism

Automatic parallelization by utilizing implicit parallelism of functional programming languages is still at an experimental stage, especially when we try to do so by applying static analysis [95]. Thus recent work on this line of research is relying on information obtained at runtime. For example, a paper on implicit parallelism in Haskell [96] achieves positive speed-up on several benchmarks by iteratively determining parallelization points based on runtime profiles. Static analysis seems

to have a limitation in utilizing implicit parallelism. Then, a naive question is: can the dynamism in GoI improve such an analysis, from a more semantic (and hopefully more fundamental) viewpoint? There in fact exists a support for this perspective: a research on implementation of an abstract machine for λ calculus, called PELCR (Parallel Environment for optimal Lambda-Calculus Reduction) [81] exists. The implementation is based on a variant of GoI (represented as a graph reduction system) and provides a parallel evaluation of λ terms. Although it is only concerned with λ calculus, there is room to extend the work, or the underlying GoI, to a calculus with more common and useful constructs.

2.2.3 Unified Approach to Process Calculi and Their Type Systems

The aim to provide a unifying theory for process calculi (or formal systems for concurrent computation in general) has been pursued by many researchers. Milner's bigraphs [72] and its variants are known to be able to encode calculi such as π-calculus or fusion calculus, as well as Petri nets. As for unification of type systems, an approach based on bigraphs is also proposed [27,28]. They aim at deriving individual type systems for process calculi from type systems for bigraphs and it is shown that one particular type system for a finite subset of π-calculus can indeed be derived in that way. However, only a few instances have been obtained since the proposal was made. Generic type system for π-calculus [54] is a type system parameterized by a subtyping relation and a notion called consistency condition. Several existing type systems, including those for guaranteeing race-freedom and deadlock-freedom, are shown as instances of the generic type system. Their approach works well for π-calculus, but extending it to other process calculi is yet to be done (it is indeed mentioned as future work in the paper). ψ-calculus [11] is a generalization of some process calculi including π-calculus. Similarly to the above-mentioned generic type system, its typed version [52] subsumes different type systems for process calculi. Again, the work covers π-like calculi, but different kinds of process calculi (ambient calculi for example) are excluded in their work. The reason why the generic type system [54] and ψ-calculus [52] do not easily extend to other calculi is that those type systems are defined over the syntax of the calculi. Our approach is expected to be syntax-free as the standard ones are so, and thus it is possible that our approach can be applied to different calculi in a uniform way.

Chapter 3

Preliminaries

3.1 Notations and General Notions

Throughout the thesis, the set of natural numbers is denoted by $\mathbb{N}$. The set of booleans is denoted by $\mathbb{B}$ (which is explicitly $\{\texttt{true}, \texttt{false}\}$). Given a set X, the set of (sub)distributions $\{\mu \colon X \to [0,1] \mid \Sigma_{x \in X}\, \mu(x) \leq 1\}$ is denoted by $\mathrm{Dist}(X)$. A distribution that maps $x_1 \mapsto p_1, \ldots, x_n \mapsto p_n$ is also written as $\{x_1^{p_1}, \ldots x_n^{p_n}\}$. Addition of two distributions and multiplication of a distribution by a real number are defined pointwise if the resulting function is again a distribution. Substitution is denoted by $\mathrm{M}\{x := \mathrm{N}\}$, meaning to simultaneously replace every occurrence of x in M by N. For a map $f \colon X \to Y$, $f[x' \mapsto y']$ denotes the map defined by $f[x' \mapsto y'](x) = f(x)$ if $x \neq x'$ and $f[x' \mapsto y'](x') = y'$.

We sometimes need the following notion of *undirected path* over a directed graph.

Definition 3.1. Let $G = (N, E)$ be a directed graph. A *undirected path* on G is a sequence $e_1 e_2 \ldots e_n$ where $e_i \in E \cup \{(a,b) \mid (b,a) \in E\}$ and satisfying $\mathrm{snd}(e_i) = \mathrm{fst}(e_{i+1})$ for any $i \in \mathbb{N}$. The pair e_i is said to be *forward* if $e_i \in E$; said to be *backward* if $e_i \in \{(a,b) \mid (b,a) \in E\}$.

Precisely speaking, we should distinguish $(a', b') \in E$ and $(a', b') \in \{(a,b) \mid (b,a) \in E\}$ when (a', b') and (b', a') are both in E; however we do not explicitly do so because such a case never appear in the thesis.

3.2 Abstract Reduction System

An *abstract reduction system* (ARS for short) is a transition system over a fixed set of elements, formally given as follows.

Definition 3.2 (Abstract Reduction System)**.** An *abstract reduction system* $(\mathrm{A}, \to)$ consists of a set A and a relation $\to \subseteq \mathrm{A} \times \mathrm{A}$.

Definition 3.3 (Notations)**.** We write $a \to b$ if $(a,b) \in \to$, and say that *a reduces to b*. When there exists no element $b \in \mathrm{A}$ satisfying $a \to b$, we write $a \not\to$ and a is said to be *in normal form* or *terminal*. We also write $a \to^k b$ if the element a reduces to b via k times of reductions: $a = a_0 \to a_1 \to \cdots \to a_k = b$. The transitive closure $a \to^+ b$ is defined by $\to^+ = \bigcup_{i \in \mathbb{N}} \to^i$. The transitive reflexive closure $\to^*$ is defined by $\to^* = \to^+ \cup \{(a,a) \mid a \in \mathrm{A}\}$.

The notion of ARS is prevalent in (theoretical) computer science. Terms of (either typed or untyped) λ calculus and β reduction (or in general a programming language and its small-step reduction) over terms form an ARS; a mathematical

notion defined as an equational system can be seen as an ARS if we regard the equations as a one-directional reduction relation; and formal proofs and cut-elimination also give rise to an ARS, which is one of the main topics of the thesis. Therefore we recall some basic notions in the section. For further information we refer to [14].

The following properties on ARSs are two main properties investigated about ARSs. Intuitively, normalization assures (may or must) convergence of the system, while confluence states that the final result (if exists) is irrelevant to the paths to reach it in the system.

Definition 3.4 (Normalization)**.** An abstract reduction system $(\mathrm{A}, \to)$ is said to be *weakly normalizing* if for any $a \in \mathrm{A}$, there exists $b \in \mathrm{A}$ satisfying $a \to^* b \not\to$. It is *strongly normalizing* or *terminating* if for any $a \in \mathrm{A}$, there does not exist an infinite sequence of reductions $a \to b_1 \to b_2 \to \ldots$.

Definition 3.5 (Confluence)**.** An abstract reduction system $(\mathrm{A}, \to)$ is said to be *confluent* if for any $a \in \mathrm{A}$, $a \to^* b_1$ and $a \to^* b_2$ implies existence of an element c satisfying $b_1 \to^* c$ and $b_2 \to^* c$.

A desirable property of ARSs is the *diamond property*. The reason of the name is clear if we depict the property as Figure 3.1.

Definition 3.6 (Diamond Property)**.** An abstract reduction system $(\mathrm{A}, \to)$ is said to satisfy the *diamond property* if for any $a \in \mathrm{A}$, $a \to b_1$ and $a \to b_2$ implies either $b_1 = b_2$ or existence of an element c satisfying $b_1 \to c$ and $b_2 \to c$.

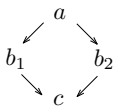


Figure 3.1: The Diamond Property

Proposition 3.7. *If an abstract reduction system satisfies the diamond property, then it is confluent.*

Proof. Since we use the proposition we explicitly show the proof. Let $a \to b_n$ and $a \to c_m$ as in Figure 3.2. The element e can be obtained by the "tiling", where each "diamond" is constructed by the diamond property. □

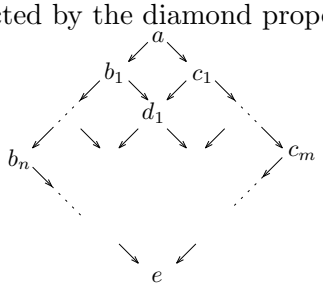


Figure 3.2: Tiling

3.3 Multiplicative Exponential Linear Logic and Proof Net

The thesis is based on the notions of *linear logic* and in particular a framework to give semantics of linear logic proofs called *Geometry of Interaction* (GoI for short). This section briefly recalls basic notions and theorems around multiplicative exponential linear logic.

Linear logic was first introduced by Girard [38] around 1987, and is often seen as a "refinement" of conventional logics such as intuitionistic logic or classical

logic, since the arrow types $A \to B$ in those logics are decomposed into the type $!\,A \multimap B$ (for intuitionistic logic) or $!\,A \multimap !\,B$ (for classical logic) using two novel primitives ! and $\multimap$. In this sense linear logic has a finer primitives than those logics. Though the introduction of linear logic was motivated from the logical viewpoint, soon after the publication of [38] it has found various usages in the realm of computer science.

3.3.1 MELL Sequent Calculus (with MIX Rule)

The contributions in the thesis are based on a fragment of linear logic called *classical multiplicative exponential linear logic*, or *MELL* for short. Given a countable set X, the *formulas* of MELL are defined by the following BNF:

$$A ::= \alpha \;\big|\; \alpha^{\perp} \;\big|\; A \otimes A \;\big|\; A \parr A \;\big|\; !\,A \;\big|\; ?\,A$$

where α is called an *atom* and $\alpha \in X$. The connectives $\otimes$, $\parr$ are called *multiplicative conjunction* and *multiplicative disjunction*, respectively, and correspond to the usual conjunction $\wedge$ and disjunction $\vee$. The ! and ? modalities (read as "bang" and "why not" respectively) in the formula $!\,A$ and $?\,A$ are called *exponential modalities*. *Linear negation* $(-)^{\perp}$ is syntactically defined as follows, by de Morgan's law

$$(A \otimes B)^{\perp} \equiv A^{\perp} \parr B^{\perp}, \quad (A \parr B)^{\perp} \equiv A^{\perp} \otimes B^{\perp}, \quad (!\,A)^{\perp} \equiv ?(A^{\perp}), \quad (?\,A)^{\perp} \equiv !(A^{\perp}).$$

Positive formulas P (resp. *negative formulas* N) are defined by the BNF $P ::= \alpha \,|\, P \otimes P$ (resp. $N ::= \alpha^{\perp} \,|\, N \parr N$). We denote a multiset of formulas by Γ or Δ and the juxtaposition Γ, Δ denotes the multiset union $\Gamma \uplus \Delta$. For a multiset $\Gamma = \{|A_1, \ldots, A_n|\}$, the notation $!\,\Gamma$ means $\{|\,!\,A_1, \ldots, !\,A_n|\}$ (similarly for the ? modality). *Linear implication* $A \multimap B$ is defined to be[1] $A^{\perp} \parr B$, which is reminiscent of logical equivalence of $A \to B = \neg A \vee B$.

A *sequent* $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ consists of multisets of MELL formulas $\{|A_1, \ldots, A_n|\}$ and $\{|B_1, \ldots, B_m|\}$ separated by the turnstile symbol $\vdash$. The rules of *MELL sequent calculus with* MIX *rule* are shown in Table 3.1. The sequent under the line in a rule is called a *conclusion* and one of the sequents over the line is called a *premise*. Like other sequent calculi, a rule $\dfrac{\Gamma \vdash \Delta}{\Gamma' \vdash \Delta'}$ means that the conclusion $\Gamma' \vdash \Delta'$ can be derived if the premise(s) $\Gamma \vdash \Delta$ are also derivable in the system. A tree constructed from the rules is called a *derivation tree* or a *proof*.

Remark 3.8. The use of multisets rather than sets is not just a design choice: $A, A \vdash B$ and $A \vdash B$ are *different* sequents and provability of either does not implies provability of the other in general. The rules like cut or $R\otimes$ indeed explicitly exhibits this character as Γ_1 and Γ_2 in the premises lead to a multiset union Γ_1, Γ_2 in the conclusion. In LK the two sequents become equivalent via weakening and contraction rules. In MELL sequent calculus, application of those rules are restricted to formulas with ! modality (LW rule and LC rule).

Remark 3.9. Usually, a sequent is defined as *sequences* of formulas and the sequent calculus system additionally has the *exchange rule* to permute the order of formulas (e.g. $B, A \vdash A, B$ can be derived from $A, B \vdash A, B$). Since the use of

[1] In *intuitionistic linear logic*, $\multimap$ is an independent connective rather than such a defined connective, simply because $\parr$ does not exist in the syntax. We focus on classical setting throughout the thesis.

$$\frac{}{A \vdash A}\ \mathsf{ax} \quad \frac{\Delta_1 \vdash \Gamma_1, A \quad \Delta_2, A \vdash \Gamma_2}{\Delta_1, \Delta_2 \vdash \Gamma_1, \Gamma_2}\ \mathsf{cut} \quad \frac{\Gamma_1 \vdash \Delta_1 \quad \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \mathsf{MIX}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \otimes B \vdash \Delta}\ L\otimes \quad \frac{\Gamma_1, \vdash \Delta_1, A \quad \Gamma_2, \vdash \Delta_2, B}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, A \otimes B}\ R\otimes$$

$$\frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \parr B}\ R\parr \quad \frac{\Gamma_1, A \vdash \Delta_1 \quad \Gamma_2, B \vdash \Delta_2}{\Gamma_1, \Gamma_2, A \parr B \vdash \Delta_1, \Delta_2}\ L\parr$$

$$\frac{\Gamma \vdash \Delta, A}{\Gamma, A^\perp \vdash \Delta}\ L\bot \quad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \Delta, A^\perp}\ R\bot$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, !\,A \vdash \Delta}\ LW \quad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, ?\,A}\ RW \quad \frac{\Gamma, !\,A, !\,A \vdash \Delta}{\Gamma, !\,A \vdash \Delta}\ LC \quad \frac{\Gamma \vdash \Delta, ?\,A, ?\,A}{\Gamma \vdash \Delta, ?\,A}\ RC$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, !\,A \vdash \Delta}\ LD \quad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, ?\,A}\ RD \quad \frac{!\Gamma, A \vdash ?\,\Delta}{!\Gamma, ?\,A \vdash ?\,\Delta}\ LP \quad \frac{!\Gamma \vdash ?\,\Delta, A}{!\Gamma \vdash ?\,\Delta, !\,A}\ RP$$

Table 3.1: MELL Sequent Calculus Rules with MIX

$$\frac{}{\vdash A, A^\perp}\ \mathsf{ax} \quad \frac{\vdash \Gamma, A \quad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}\ \mathsf{cut} \quad \frac{\vdash \Gamma \quad \vdash \Delta}{\vdash \Gamma, \Delta}\ \mathsf{MIX}$$

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

$$\frac{\vdash \Gamma}{\vdash \Gamma, ?\,A}\ ?W \quad \frac{\vdash \Gamma, ?\,A, ?\,A}{\vdash \Gamma, ?\,A}\ ?C \quad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?\,A}\ ?D \quad \frac{\vdash ?\,\Gamma, A}{\vdash ?\,\Gamma, !\,A}\ !$$

Table 3.2: Right-Hand Side Only MELL Sequent Calculus Rules

the exchange rule adds inessential technicality without benefits (at least for our purpose in the thesis), we choose the definition by multisets that is essentially the same and simpler.

Remark 3.10. By repeatedly applying $R\bot$ rule, any sequent $\Gamma \vdash \Delta$ can be turned into the form $\vdash \Gamma^\perp, \Delta$. All rules acting on the left-hand side of sequents commute with this procedure: the result of applying a left-hand side rule (e.g. $L\otimes$) and then applying the procedure is the same as the one obtained by applying the procedure first and then applying a corresponding rule on the left-hand side (e.g. $R\parr$). Thus the calculus can be defined by the right-hand side rules only as shown in Table 3.2, without changing expressivity. We use the right-hand side only rules in the next subsection since correspondence with proof structures is clearer.

3.3.2 MELL Proof Net

In the previous section we saw the rules of MELL sequent calculus. Linear logic has another convenient graphical syntax to derive valid formulas, called *proof nets* [38]. An advantage is that inessential syntactic differences in sequent calculus disappear in proof nets. Proof nets will be defined as a subset of *proof structures* defined as follows:

Definition 3.11 (Nodes and Notations)**.** The set of *nodes* of MELL proof structures is $\{\mathsf{ax}, \mathsf{cut}, \otimes, \parr, ?\mathsf{c}, ?\mathsf{w}, ?\mathsf{d}, !\}$ shown in Figure 3.3. Each node except the ! node has a fixed number of incoming edges and outgoing edges labeled by MELL formulas. An incoming (resp. outgoing) edge of a node is called a *premise* (resp. *conclusion*) of the node. An edge labeled by a formula A is said to be *of type*

A. We depict a graph consisting of those nodes directed from up to bottom: premises (resp. conclusions) of a node is drawn as an edge coming from above (resp. below) the node. For example, an ax node has no premises and two conclusions of type A and $A^\perp$ respectively, and a $\otimes$ node has two premises of type A, B and one conclusion of type $A \otimes B$. A ! node has a conclusion (called the *principal conclusion* of the ! node) of type $!\,A$ and n-many conclusions (called *auxiliary conclusions* of the node) of type $?\,A_1, ?\,A_2, \ldots, ?\,A_n$ for some $n \in \mathbb{N}$. Note that n can possibly be 0.

Definition 3.12 (MELL Proof Structure)**.** An *MELL proof structure* is a finite directed graph built from the nodes in Figure 3.3 where

- each edge is equipped with a type that matches with the types specified by the nodes it is connected to;
- some edges may be dangling, i.e. may not be connected to any node. Those dangling edges are called the *conclusions of the structure*. By abuse of notation, their types are also called the conclusions;
- the graph is equipped with a total map from all the ! nodes in it to MELL proof structures called the *contents* of the ! nodes, satisfying that the types of the conclusions of a ! node coincide with those of its content.

The *depth* of a node in a structure is defined by: if a node is not in a content of a box, the depth of the node is defined to be 0 (the node is said to be *at depth* 0 or *at surface*); otherwise the depth of the node is $n + 1$ (said to be *at depth* $n + 1$), where n is the depth of the ! node that has the node as its content. The graph consisting of the nodes at depth 0 is called the *surface structure*.

Although formally a ! node and its content are not connected as a directed graph, it is intuitive and convenient to draw them as if the ! node is a "box" filled with its content, as shown in Figure 3.4. For this reason a ! node is also called a !-*box*.

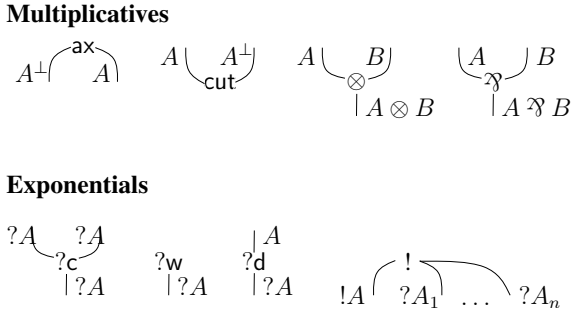


Figure 3.3: MELL Nodes

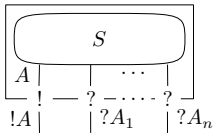


Figure 3.4: !-Box

The proof nets are equipped with the notion of *reduction* that can be seen as a procedure of cut-elimination written in the language of graph rewriting.

Definition 3.13. *Reduction* of MELL proof structures is defined by the reduction rules shown in Figure 3.5. The reduction rules are defined to be *local*: a redex of a reduction rule in a proof structure rewrites to the reduct of the rule and the rest of the proof structure remains the same.

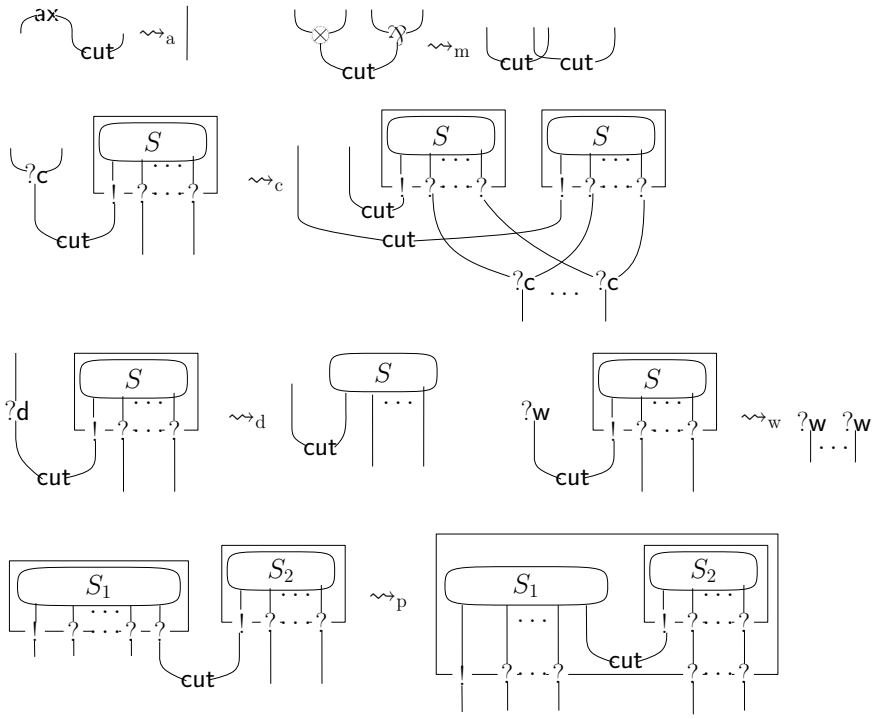


Figure 3.5: Reduction Rules of MELL Proof Structure

As expected from the names of the nodes, there exists a correspondence between (derivation trees of) MELL logical formula and MELL proof structures as shown in Figure 3.6 and 3.7. However, not every proof structure corresponds to a derivation tree.
The characterization of those structures that have a derivation of an MELL formula is called a *correctness criterion*; many such criteria are known [10], and among them the most common one is the one by Danos and Regnier [22]. The following is a minor variant of theirs.

Definition 3.14 (Switching Path)**.** Given an MELL proof structure R, a *switching path* is an undirected path on R (i.e. a path allowed to traverse a directed edge forward or backward) satisfying that

- on each $\parr$ node, the path uses at most one of the premises;
- on each $?c$ node, the path uses at most one of the premises;

and the path does not use any edge twice or more.

Definition 3.15 (Correctness Criterion)**.** The *correctness criterion* states that given an MELL proof structure R, all of its switching paths are acyclic. Such a path is said to be *correct*.

Remark 3.16. Precisely speaking, there are two differences from the correctness criterion in [22]:

- In [22], correctness is defined by *correctness graphs* rather than switching paths. As for acyclicity the two definitions can easily be shown to be equivalent. We employ the definition by paths here and later in the thesis because it is useful in some proofs.
- We drop the connectedness condition in [22]. In the terminology of linear logic, this corresponds to allowing MIX rule.

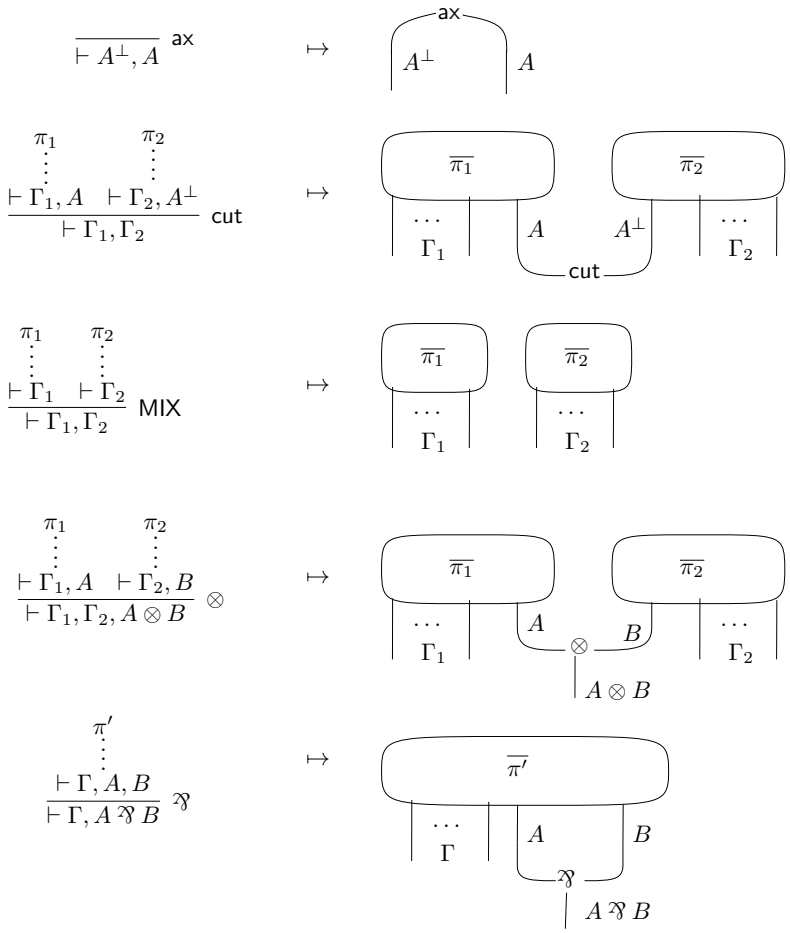


Figure 3.6: Correspondence between Sequent Calculus Proof and Proof Structure (1)

Definition 3.17 (MELL Proof Net)**.** An *MELL proof net* is an MELL proof structure of which surface structure satisfies the correctness criterion and the content of every box in it is itself an MELL proof net.

Lemma 3.18. *Let R be an MELL proof net. If $R \rightsquigarrow S$ then the MELL proof structure S also satisfies the correctness criterion, hence S is an MELL proof net.*

The fact that the correctness criterion is indeed a characterization of those "correct" structures is verified by the next theorem. Together with Lemma 3.18, reduction of proof nets indeed serves as cut-elimination procedure.

Theorem 3.19 (Sequentialization Theorem)**.** *Given an MELL proof structure R, if it is an MELL proof net (i.e. satisfies Danos-Regnier correctness criterion) then there exists a derivation tree π of MELL sequent calculus satisfying $\overline{\pi} = R$.*

3.4 Geometry of Interaction as Token Machine

The *Geometry of Interaction*, proposed by Girard himself in [39] is a framework to give a semantics of linear logic proofs (or proof systems derived from linear logic) as "consistent paths" in proofs in the system. Intuition is that such a path in a proof represents an information flow over the proof that seeks the source of an atom (thought as a data) in a logical formula.

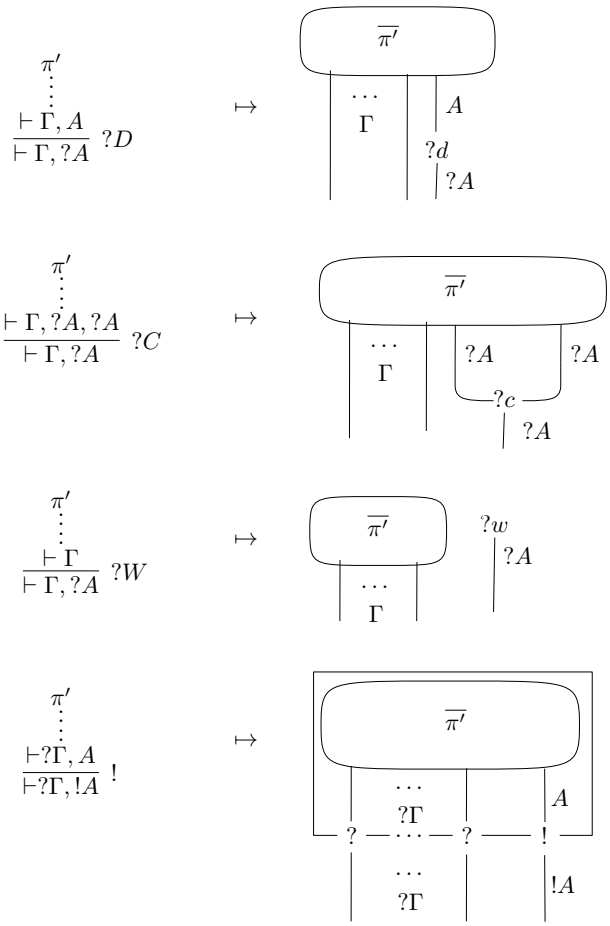


Figure 3.7: Correspondence between Sequent Calculus Proof and Proof Structure (2)

There are various representation of GoI. We follow the most concrete one, namely the *token machine*, or specifically the *Interaction Abstract Machine* (IAM) given by Danos and Regnier [23].

Definition 3.20. A *stack* s is given by the following BNF:

$$\sigma ::= * \mid l(\sigma) \mid r(\sigma) \mid \lceil \sigma, \sigma \rceil \quad ,$$

$$\mathrm{s} ::= \epsilon \mid l.\mathrm{s} \mid r.\mathrm{s} \mid \sigma.\mathrm{s} \quad ,$$

where ϵ denotes an empty stack and the dot (.) denotes concatenation. Concatenation $\mathrm{s}.\epsilon$ or $\epsilon.\mathrm{s}$ with an empty stack is defined to be s.

Definition 3.21 (Occurrence Indication)**.** Let A be an MELL formula. A stack s *indicates* an occurrence of an atom α in A if $\mathrm{s}[A] = \alpha$ holds, where $\mathrm{s}[A]$ is inductively defined by

- $\epsilon[\alpha] = \alpha$,
- $l.\mathrm{s}[A \square B] = \mathrm{s}[A]$;
- $r.\mathrm{s}[A \square B] = \mathrm{s}[B]$;
- $\sigma.\mathrm{s}[\spadesuit A] = \mathrm{s}[A]$;
- $\mathrm{s}[A]$ is undefined otherwise;

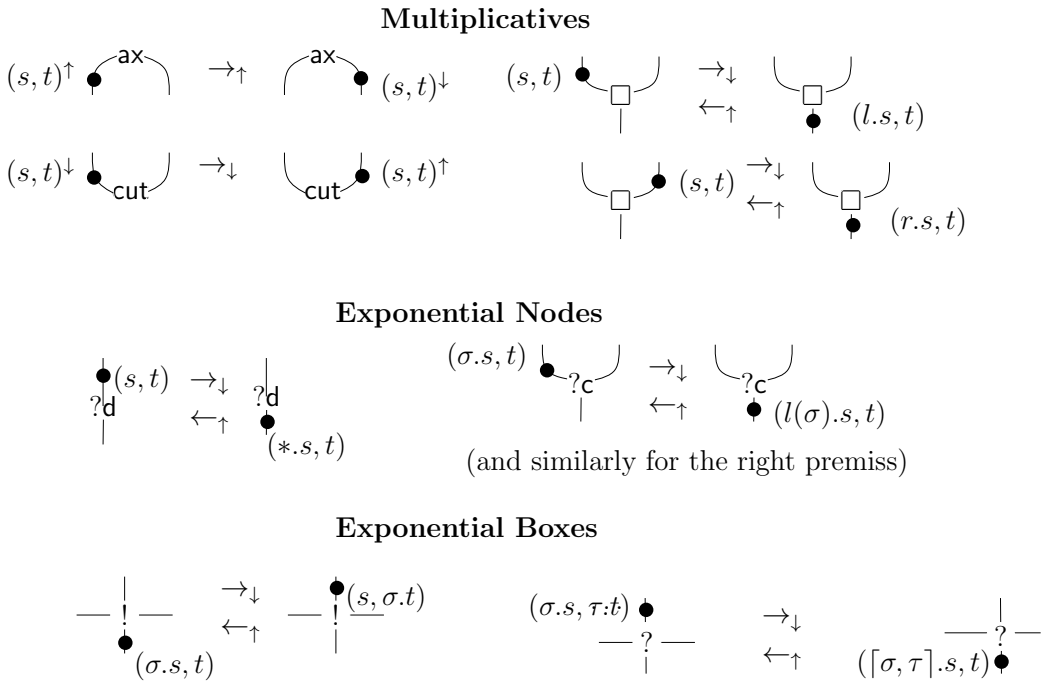


Figure 3.8: IAM Transition Rules

where $\square \in \{\otimes, \parr\}$.

Definition 3.22. Let R be an MELL proof net. A *state* is a triple $(e, \mathrm{s}, \mathrm{t})$ where e is an edge of R (possibly of the content of a box), s is a stack indicating an atom in the type A of the edge, and t is a stack.

Intuitively, the token machine we are going to define employs a single *token* running over a proof net. A state $(e, \mathrm{s}, \mathrm{t})$ represents a token on the edge e, keeping track of an atom in the type of e by the stack s. The other t remembers in which copy of ! box the token lies.

Definition 3.23 (Initial and Final State)**.** Let R be an MELL proof net. An *initial state* is a state $(e, \mathrm{s}, \epsilon)$ where e is a conclusion edge of R, s indicates a negative atom in the type A of the edge e. Similarly, a *final state* is a state $(e, \mathrm{s}, \epsilon)$ where e is a conclusion edge of R, s indicates a positive atom in the type A of the edge e.

Definition 3.24 (Transition Rules)**.** The *transition rules* over states are pictorially defined in Figure 3.8, by representing a state $(e, \mathrm{s}, \mathrm{t})$ as a bullet and a pair (s, t) next to the edge e. An arrow with $\rightarrow^{\uparrow}$ (resp. $\rightarrow^{\downarrow}$) can be applied only when s indicates a negative (resp. positive) atom. The symbol $\square$ in the figure represents a $\otimes$ node or a $\parr$ node.

Thus, intuitively a token starts from a conclusion of a given net, travels over the net, and ends its travel again a conclusion. The states and the transitions defined above form a transition system:

Definition 3.25 (Interaction Abstract Machine)**.** Given an MELL proof net R, the *Interaction Abstract Machine (IAM)* $\mathcal{M}_R$ is the transition system $(\mathcal{S}, \rightarrow)$ where $\mathcal{S}$ is the set of all states and $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the one induced by the transition rules.

Via the IAM, we can give a semantics of proof nets. It is a semantics in the sense that it is invariant under reduction of proof nets, with a condition that any

? modality appears in the conclusions (Theorem 3.27). In other words, we can calculate the result of proof net reduction by the IAM instead of reducing a given net.

Definition 3.26 (Token Machine Semantics)**.** Given an MELL proof net R, the *token machine semantics* $[\![R]\!]$ of the proof net is a partial function $\mathrm{Init}_R \rightharpoonup \mathrm{Fin}_R$ given by $[\![R]\!](\mathrm{T}) = \mathrm{U}$ if the run of the IAM $\mathcal{M}_R$ from the initial state T terminates in the final state U, and otherwise undefined.

Theorem 3.27. *Let R be an MELL proof net without any ? in its conclusion. If $R \rightsquigarrow S$, then $[\![R]\!] = [\![S]\!]$.*

3.5 Quantum Computation

In the thesis we do not investigate quantum computation itself, but rather we would like to show that the minimalistic requirement to constitute quantum computation is accommodated in our framework in Chapter 5. Therefore, we base our argument on a setting as simple as possible: we only consider finite-dimensional case, and thus we use vectors in a finite-dimensional Hilbert space to represent a quantum state. For further details see the standard literature [77].

A *quantum bit* (often called *qubit*) is represented by a non-zero vector in a two-dimensional complex Hilbert space. The orthonormal basis is written as $|0\rangle$ and $|1\rangle$. Thus a qubit is represented by a *superposition* of $|0\rangle$ and $|1\rangle$. Then a state of a multi-qubit system is represented by the tensor product of states of qubits. For example, the basis of a 2-qubit system consists of tensor products of basis vectors of the two 1-qubit system, that is $|0\rangle \otimes |0\rangle, |0\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle$, and $|1\rangle \otimes |1\rangle$. We write $|00\rangle$ for $|0\rangle \otimes |0\rangle$, and so on. In such a multi-qubit system, a peculiar state can exist: For example, the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is one of such: as easily calculated, the state cannot be decomposed into a tensor product $|\varphi\rangle \otimes |\psi\rangle$ of two 1-qubit states φ and ψ. Such a state is called an *entangled state*, and is the computational power of quantum computation arises from such entangled states.

Two primitives can be applied to a quantum state: one is application of *unitary gates* represented by unitary maps in the Hilbert space. Among unitary gates, H (*Hadamard gate*) that sends $|0\rangle$ to $\frac{\sqrt{2}}{2}(|0\rangle + |1\rangle)$ and $|1\rangle$ to $\frac{\sqrt{2}}{2}(|0\rangle - |1\rangle)$, and CNOT (*controlled not gate*) that sends $|xy\rangle$ to $|x \oplus y\rangle \otimes |y\rangle$ (here $\oplus$ is the classical exclusive or) are particularly important, since by applying $\mathsf{CNOT}(\mathsf{H}|0\rangle \otimes |0\rangle)$ we can obtain an entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

The other is *measurement*: when we would like to retrieve classical information (bits) from a quantum state in superposition, in general application of measurement *probabilistically* yields 1 or 0, with probabilities coming from the (normalized) coefficients. Concretely, given a quantum state $\alpha_0|0\rangle \otimes \phi_0 + \alpha_1|1\rangle \otimes \phi_1$, measurement on the first qubit yields 0 with probability $|\alpha_0|^2$, resulting a state $|0\rangle \otimes \phi_0$, or 1 with probability $|\alpha_1|^2$ resulting $|1\rangle \otimes \phi_1$. Therefore, measurement has another distinctive feature: after measurement, the original state globally *collapses* into one of the two possible results.

3.6 Nominal Set

The notion of *nominal set* has been devised to mathematically cleanly handle the names appearing in computer science, especially in the study of programming languages. Although it is a theory as rich as yielding a book [84], what we need in the thesis is only the very basic notions.

Definition 3.28. Let G be a group. A *G-set* $(M, \cdot)$ is a set M equipped with an action of G on M, i.e. a binary operation $(\cdot)\colon G \times M \to M$ that respects the group operation, meaning $g \cdot (g' \cdot m) = (g \cdot g') \cdot m$ for each $g, g' \in G$ and $m \in M$.

Definition 3.29. Let I be a countably infinite set; let M be a set equipped with an action of the group $\mathrm{FinBij}(I)$ of *finitary permutations* of I.

A *support* for $m \in M$ is a subset $A \subseteq I$ such that for all $\sigma \in \mathrm{FinBij}(I)$, $\forall i \in A, \sigma i = i$ implies $\sigma \cdot m = m$.

A *nominal set* is a $\mathrm{FinBij}(I)$-set all of whose elements have finite support. In this case, if $m \in M$, we write $\mathrm{supp}(m)$ for the smallest support of m.

The complementary notion of support is freshness: $i \in I$ is *fresh* for $m \in M$ if $i \notin \mathrm{supp}(m)$.

We write $(i\ j)$ for the transposition which swaps i and j.

We will make use of the following characterization of support in terms of transpositions:

Proposition 3.30. *$A \subseteq I$ supports $m \in M$ if and only if for every $i, j \in I - A$ it holds that $(i\ j) \cdot m = m$. As a consequence, for all $i, j \in I$, if they are fresh for $m \in M$ then $(i\ j) \cdot m = m$.*

Chapter 4

Synchronous Interaction Abstract Machine

4.1 Motivation

This chapter introduces the main idea and contribution of the thesis, namely a notion of *multi-token machine*, called the *Synchronous Interaction Abstract Machine (SIAM)*. We are going to develop various notions around the multi-token machine and its underlying proof net system. Before doing so, we describe the original motivation of the entire work of the thesis. The section is devoted to this purpose.

The notion of multi-token machine in the thesis was first initiated in [99] for a multiplicative (thus very simple, restricted) fragment, and later developed further in [18] by accommodating exponentials and recursion. The technical details in this chapter are mostly those of [18], with some modification where needed. Among the content, the proof theory on the nets partly owes to our coauthor Claudia Faggian; most of the theory on the multi-token machine, especially its details, has been developed by the author on the other hand.

4.1.1 Deterministic, Arithmetic Case

Let us first recall the GoI token machine by Mackie [69]. A token bears a pair (s, n) where s is the information required to find its path; n is a query for a natural number or an actual natural number under calculation. The primitives under consideration in the paper include constants $\overline{n}$ (0-ary), the successor and the predecessor (both unary), as in the usual PCF. Perhaps it might seem that we already need multiple tokens if we have a binary (or in general n-ary for $n > 1$) primitive such as the addition operator $+$; however, the case is still feasible with a single token by stipulating a few more transition rules. Let us take a PCF term $4 + (\mathtt{succ}\, 1)$ as an example. As a natural extension of the style Mackie took to accommodate `succ` node in the paper [69], we give the interpretation of a binary function $+$ as a node with three conclusions, one with type $\mathbb{N}$ and the others with $\mathbb{N}^{\perp}$, as shown in Figure 4.1. Then, the net corresponding to the term M is as in Figure 4.2. To extend the token machine semantics so as to interpret

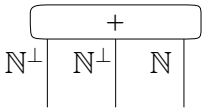


Figure 4.1: The Node for +

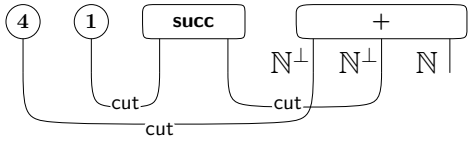


Figure 4.2: A Net Interpreting the PCF Term

the operator $+$, we add two more symbols $+_1, +_2$ to the language of stacks in the token machine. Then, a token can correctly calculate the natural number

that coincides with the normal form of the term M, namely 6 in this case, by following the rules shown in Figure 4.3. Intuitively, the first rule throws a query

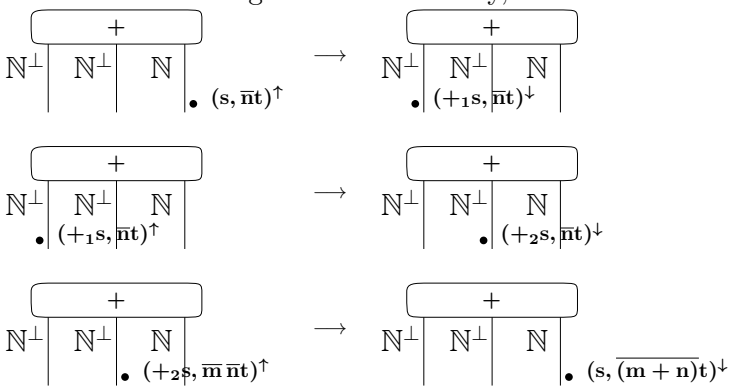


Figure 4.3: Additional Transition Rules for Summation

for the first argument of the + node; the second one throws a query for the second argument, after receiving a natural number for the first query; and the last one means that both of the two queries successfully returned, thus the rule adds the two returned values and returns their sum as the final result. Indeed, these rules are essentially the same as the categorical token machine used by Hasuo and Hoshino [45]. Several snapshots of the whole execution is described in Figure 4.4.

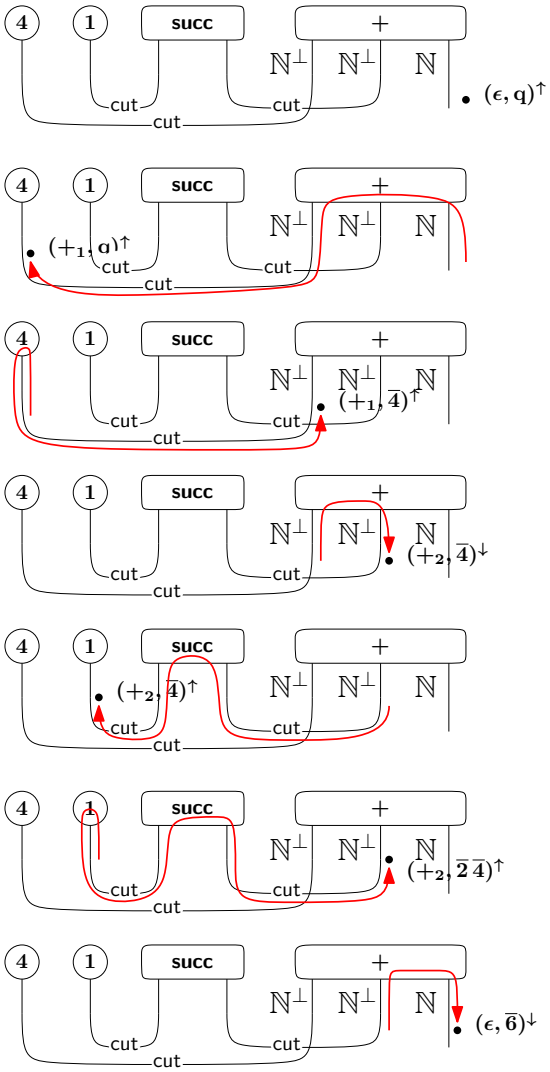


Figure 4.4: Execution

4.1.2 Quantum Case

The situation fundamentally changes when we start to consider a *quantum* language, and it is indeed our original motivation to introduce the multi-token machine framework. As briefly explained in Section 3.5, one of the peculiar phenomena in quantum computation is *entanglement*: there exist quantum states that cannot be decomposed into a tensor product of two states. We are going to look at two examples involving entanglement. One is the original leading example that motivated us; however later we realized that it can be interpreted by a single-token framework with some more complication. The other is more fundamental, and as for this situation we believe that it exhibits a clear motivation for introducing a token machine with multiple tokens. Let us look at the first one: we would like to express a term $\mathsf{let}\ \langle x, y\rangle = (\mathsf{CNOT}(\mathsf{H}\,\mathsf{new} \otimes \mathsf{new}))\mathsf{in}\, x \otimes (\mathsf{CNOT}(\mathsf{H}\mathsf{new} \otimes y))$ with quantum primitives in an appropriately extended net, where new represents preparation of a new qubit; H and CNOT are the 2-qubit Hadamard gate and the CNOT gate, respectively. To exploit the advantage of being based on linear logic, it seems natural to assume that the tensor product $\otimes$ in the quantum language is interpreted by the $\otimes$ connective in linear logic[1].In the same way as the classical case above, the most natural candidates for the nodes for qubit preparation and a unitary gate on n qubits are the ones in Figure 4.5. Using those nodes, the

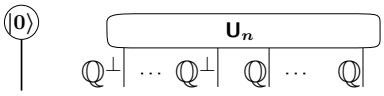


Figure 4.5: Nodes for Quantum Primitives

term may be translated into the net depicted in Figure 4.6.

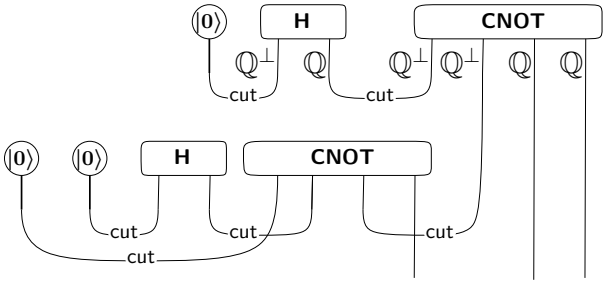


Figure 4.6: A Net Interpreting the Term

As a trial, let a token start traveling from the leftmost conclusion as in Figure 4.7. It first hits CNOT; then as in classical case it starts to search for the first argument of the CNOT node, and hits the $|0\rangle$ node. Then it returns to the CNOT and searches for the second argument, hits H and again searches for its argument, hits another $|0\rangle$. Finally return to the CNOT node and tries to return the state of the second qubit of the 2-qubit state—which fails and gets stuck, because it is impossible to represent "the state of the second qubit" of the entangled state $\mathsf{CNOT}(\mathsf{H}\,\mathsf{new} \otimes \mathsf{new})$.

Though the execution can be modified not to get stuck if we separate the quantum data from the token and instead make it bear a *pointer* to the separated data; this is indeed what the categorical token machine [45] essentially does. However, we still face another mismatch between the term and the (prospected) interpretation as a token machine, which the following example run of the machine shows. To see it, let a token again start from the leftmost conclusion. Since we now do not require a token to retrieve a one-qubit state explicitly,

[1]This is not the case in Hasuo and Hoshino [45]. There are two kinds of tensor products in their type system, one for "not entangled for sure" products and another for "maybe entangled" products, and only the former is interpreted by the multiplicative conjunction $\otimes$.

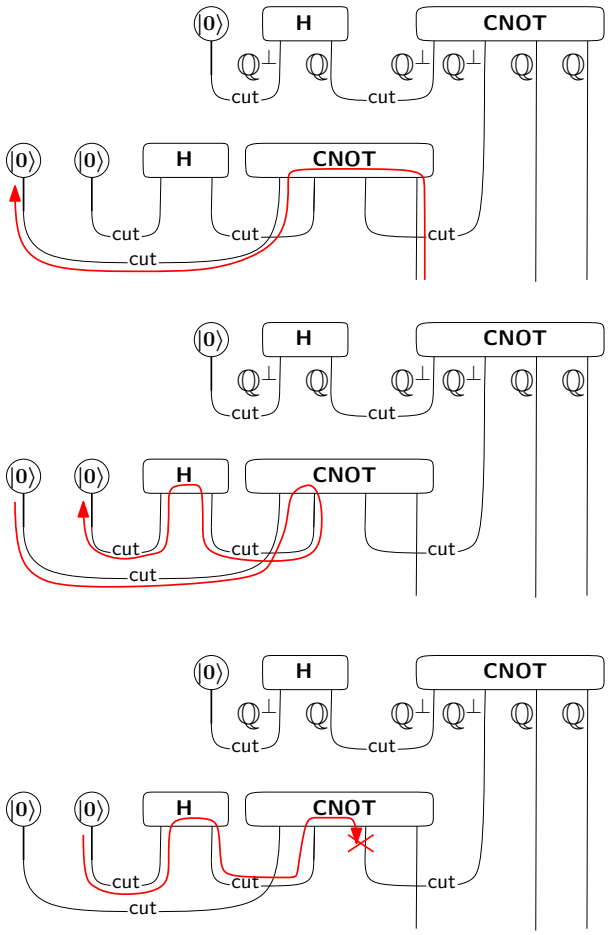


Figure 4.7: Execution (That Fails)

it may travel around the leftmost CNOT node and safely return to the original conclusion—with the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ calculated along the path, and a pointer to the second qubit of the state. The point is that the obtained state is *not* the intended one ($\frac{1}{2}(|000\rangle + |011\rangle + |101\rangle + |110\rangle)$, where the conclusion we chose corresponds to the first qubit), and even not compatible with the intended one, due to entanglement. The reason why the state obtained by this single-token machine does not coincide with the desired one is that the path of a single token may miss a (possibly large) part of the net that is actually critical to calculate the resulting entire state. The behavior is well known as the fact that typical GoI models are only adequate for call-by-name calculi, not for call-by-value ones; this nature is also influencing here, in a harmful way.

Hence, if we aim to model quantum computation in the framework of GoI token machine, it only makes sense to compute all the qubits *at the same time*. Considering that one token is in charge of one base type in standard token machines, it implies that we have *multiple* tokens at the same time. In this way, the main theme of the thesis arises. The formal definition and properties of such multi-token machines will be introduced in Section 4.2 and thereafter.

4.2 SMEYLL Proof Nets

To define multi-token machines as informally described in the previous section, we first of all need their underlying proof nets, called *SMEYLL proof nets* (meant to be read like "smile nets", and to be the initials of Linear Logic with Synchro-

nization, Multiplicatives, Exponentials, and Y-combinator). The section consists of the definition of SMEYLL proof nets.

4.2.1 Formula

The *logical formulas* are the same as (classical) MELL formulas with units:

$$A ::= 1 \;\big|\; \bot \;\big|\; A \otimes A \;\big|\; A \parr A \;\big|\; !A \;\big|\; ?A.$$

Following the usual convention in linear logic, linear negation $(-)^\bot$ is defined on any formula. The linear implication $A \multimap B$ is defined to be $A^\bot \parr B$. The *positive* (resp. *negative*) types are defined by $P ::= 1 \;\big|\; P \otimes P$ (resp. $N ::= \bot \;\big|\; N \otimes N$). These formulas will be equipped to proof nets we will define later in the chapter.

4.2.2 Proof Structures

As in linear logic, *SMEYLL proof nets* (or shortly *SMEYLL nets*) will be defined to be *proof structures* satisfying a *correctness criterion* akin to Danos & Regnier's acyclicity condition [22][2]. An interesting point is that our correctness criterion possesses an additional meaning in the theory of our multi-token machines (that will be introduced in Section 4.3), that is, it can be seen as a sufficient condition for *deadlock-freedom* of the machines.

The kinds of *nodes* (also called *links* or *cells* in the literature) used to construct SMEYLL proof structures are shown in Figure 6.1. The edges are directed from top to bottom: an incoming edge of a node is drawn as an edge coming from above the node, and an outgoing edge is drawn as an edge from below. The incoming (resp. outgoing) edges of a node are called *premises* (resp. *conclusions*) of the node. Among them, what we introduce are three kinds of nodes specific to our structures: *sync, $\bot$-box, and Y-box*. The other nodes (multiplicatives, units, exponentials) are exactly the same as those of the MELL proof nets (see Section 3.3).

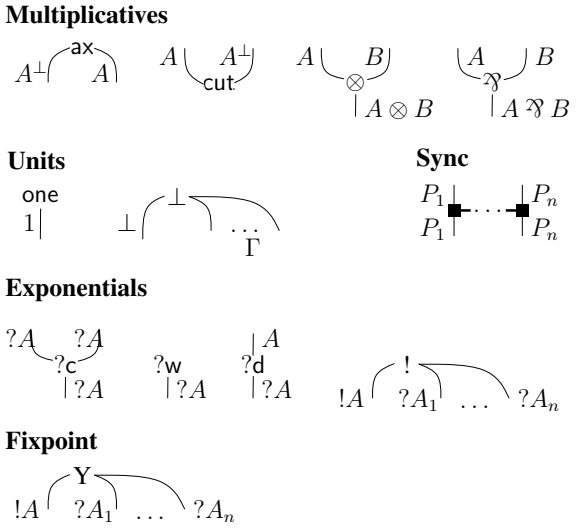


Figure 4.8: SMEYLL Nodes

- A sync node has n-many pairs of a premise and a conclusion, pairwisely having the same positive type. As shown in Figure 6.1, a sync node is

[2] As mentioned in Chapter 3, hereafter in the thesis we drop the connectedness condition that is present in [22]. This corresponds to allowing the MIX rule that makes definition of proof nets with exponential rules easier.

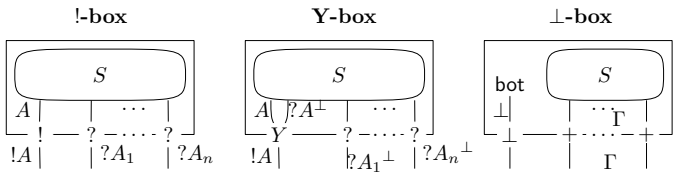


Figure 4.9: !-, Y-, and ⊥-Boxes, Pictorially

depicted as n-many black squares connected by a line (note that it is a single node, not n-many nodes connected by edges). It plays a crucial role to make multiple tokens *synchronize* in Section 4.3.

- A ⊥-box is a box, like a !-box in the usual MELL nets, with a principal conclusion typed by ⊥ and auxiliary conclusions typed by $\Gamma = A_1, A_2, \cdots, A_n$ (possibly $n = 0$). A ⊥-box must be associated with a content with conclusions Γ and ⊥ *node*, another new kind of node with no premises and one conclusion typed with ⊥. This corresponds to adding a rule

$$\frac{\vdash \Gamma}{\vdash \bot, \Gamma}$$

  for the unit ⊥ to the sequent calculus.

- A Y-box is also a box, with a principal conclusion typed by $!\,A$ and auxiliary conclusions typed by $\Gamma = A_1, A_2, \cdots, A_n$ (possibly $n = 0$). A Y-box has one content with conclusions A and $?\,A^{\perp}$ that is intuitively a *recursive function*. This corresponds to adding a "fixpoint" rule

$$\frac{\vdash A, ?\,A^{\perp}, ?\,\Gamma}{\vdash !\,A, ?\,\Gamma}$$

  to the sequent calculus.

The others are all identical to the nodes for MELL proof structures with multiplicative units.

Remark 4.1. In [99], the authors introduced nodes for *unitary gates* and *measurement* to interpret a variation of quantum λ calculus: the former ones have essentially the same typing as our sync nodes, and the latter ones correspond to multi ⊥-boxes. However, besides the calculus in [99] does not accommodate exponentials or recursion, the choice taken there actually obfuscates the precise roles of *structures* and *computational effects*, and both the expressiveness and the properties the proof nets satisfy are weaker than the ones in this thesis. By separating the structures (as sync nodes and multi ⊥-box) from effects (will be added in the later in this thesis), we obtain a clearer and more modularized tool to analyze computation in general.

Remark 4.2. A node for recursion itself is not a novel notion: Montelatici [75] already introduced such one, though with a slightly different typing rule and a reduction rule. Indeed, if the formula A is negative, our Y-box can be seen as a composition of the usual !-box and Montelatici's Y-box. Thus our construction can in turn be seen as a relaxation of Montelatici's Y-box to not necessarily polarized formulas. This relaxation is crucial when we interpret programming languages later (Section 4.5), since an interpretation of a function type is neither positive nor negative in general. Moreover, in [75] no Geometry of Interaction model is given for the nets, while we have a concrete one that will be also introduced later in this chapter (Section 4.3).

Definition 4.3 (SMEYLL Proof Structure)**.** A *SMEYLL proof structure* is a finite directed graph built from the nodes in Figure 6.1 where

- each edge is equipped with a type. Each node has to satisfy its constraint on the types of premises and conclusions as shown in Figure 6.1;
- some edges may be dangling, i.e. may not be connected to any node. Those edges are called the *conclusions* of the structure. By abuse of notation, their types are also called the conclusions;
- the graph is equipped with a total map from all the boxes (!-boxes, Y-boxes, and multi $\bot$-boxes) in it to their contents that are again proof structures.

The *depth* of a node in a structure is also defined similarly to the MELL case.

Like the MELL case, it is intuitive and convenient to draw a box and its content(s) as if it is really a "box" filled with its content (Figure 4.9).

Example 4.4. An example of SMEYLL proof structure is shown in Figure 4.10. Observe that all the typing of edges are instances of the typing in Figure 6.1.

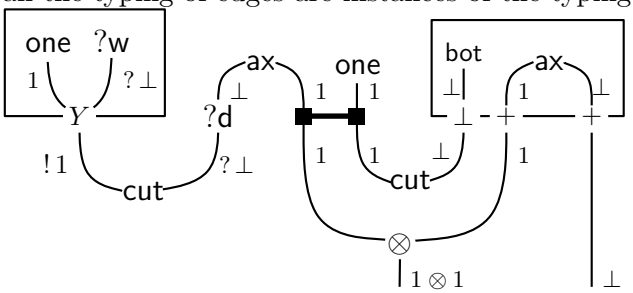


Figure 4.10: Example of SMEYLL Proof Structure

4.2.3 Reduction Rules

The *reduction rules* $\rightsquigarrow$ for the SMEYLL proof structures are also an extension of those for MELL, except we require two constraints on reductions:

Definition 4.5 (Reduction Rules of SMEYLL Proof Structure)**.** The *reduction rules* of SMEYLL proof structures are shown in Figure 4.11. They are local by definition: any reduction rewrites the redex into the reduct of the rule applied and keeps the rest of the proof structure unchanged. Each rule is named for convenience.

Moreover, we require every reduction to be *surface*, meaning that we restrict any reduction to be at depth 0, and to be *closed*, meaning that reductions on boxes are prohibited if the box of which the principal conclusion in the redex has an auxiliary conclusion.

We write $\rightsquigarrow^*$ for the reflexive and transitive closure of the reduction relation $\rightsquigarrow$ induced by the reduction rules. We also write $R \not\rightsquigarrow$ if the structure R is *normal*, i.e. if there does not exist any structure S that satisfies $R \rightsquigarrow S$.

The rules on MELL connectives in Figure 4.11 are basically identical to MELL proof reduction rules. Each of the newly-introduced nodes comes with new reduction rules; the intuition of the rules for newly-introduced nodes in Figure 4.11 are as follows.

- The rules on sync nodes first "push up" the nodes ($\rightsquigarrow_{\text{s.com}}$ rule); if a sync node reaches the "top" of the structure with all the premises connected to unit nodes, then it vanishes in a synchronous way ($\rightsquigarrow_{\text{s.el}}$ rule).

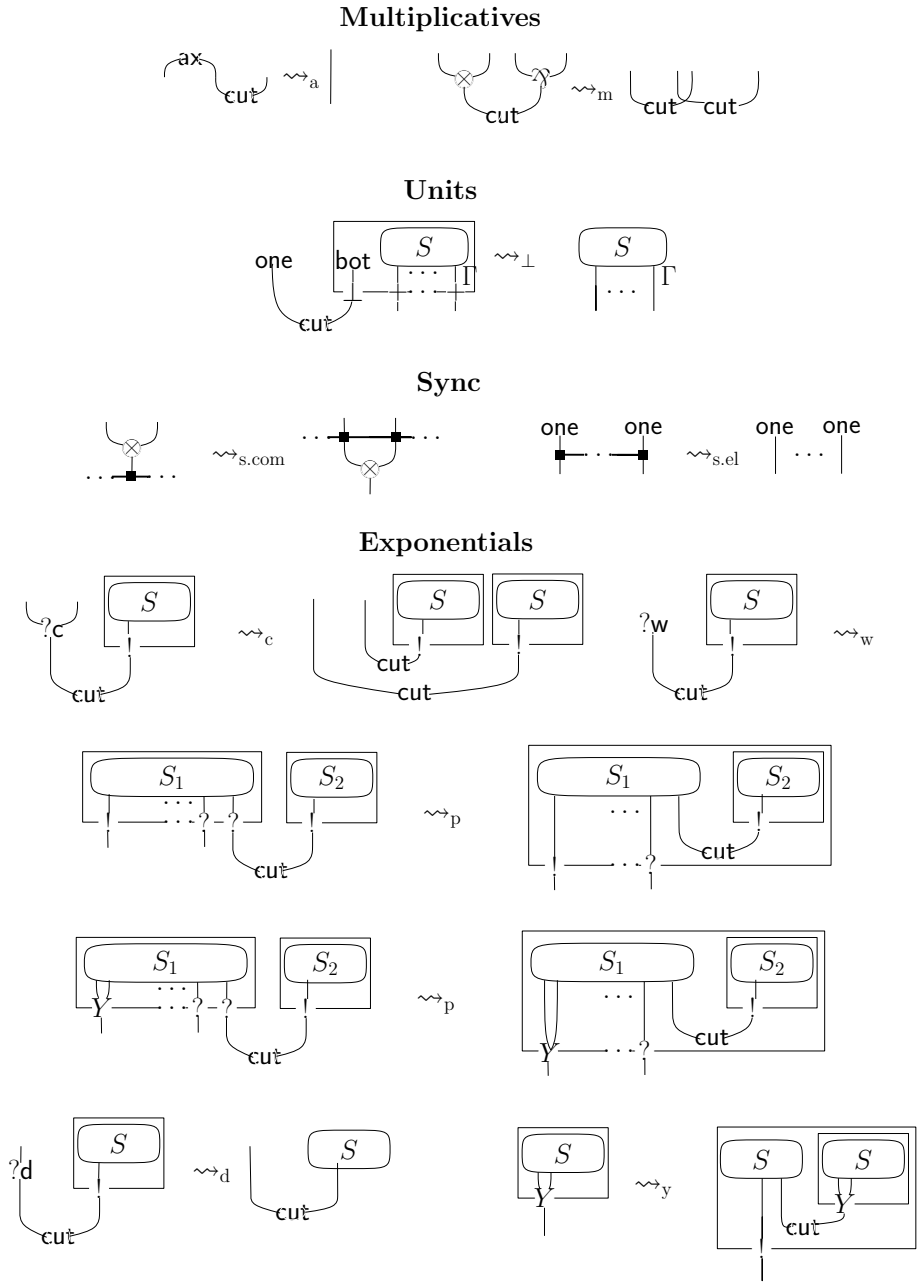


Figure 4.11: Reduction Rules

- The rule $\rightsquigarrow_{\bot}$ for $\bot$-boxes "opens" a $\bot$-box, which makes the content of the box ready to be reduced (the content cannot be reduced unless it comes out to surface because we require surface reduction).
- A Y-box turns into a ! box by $\rightsquigarrow_{\mathrm{y}}$ rule, unfolding its content and at the same time duplicating itself. The intention becomes clear by looking at the translation of a PCF term $\texttt{letrec}\, f\, x = M\, \texttt{in}\, N$, shown in Figure 4.12. The structure reduces to the translation of explicit substitution of f by $\lambda x.\texttt{letrec}\, f\, x = M\, \texttt{in}\, M$ in N. Note that the duplicated substructure is in the inner box, thus due to surface reduction it temporally stops to reduce further, and hence the rule itself does not immediately introduce divergence.

Surface reduction prohibits any reduction inside a box; we require it in order to interpret call-by-name calculus with effects appropriately. The role of closed reduction is simplification; for *simple* nets (a notion defined later) we can always

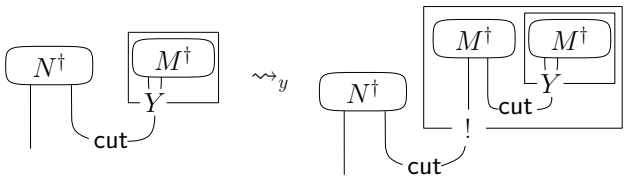


Figure 4.12: Example of Y reduction

find a closed redex, thus it is sufficient to analyze closed reductions only.

Example 4.6. An example of SMEYLL proof structure is shown in Figure 4.13. There are three cuts (named **A**, **B**, and **C**, shown in red) in the structure. The cut **A** cannot be reduced since it is not closed (the box in the redex has an auxiliary conclusion). The cut **B** cannot reduced neither, since it is not surface (it is in a !-box). Only **C** can be reduced: the box whose principal conclusion is involved in the redex is surface and closed.

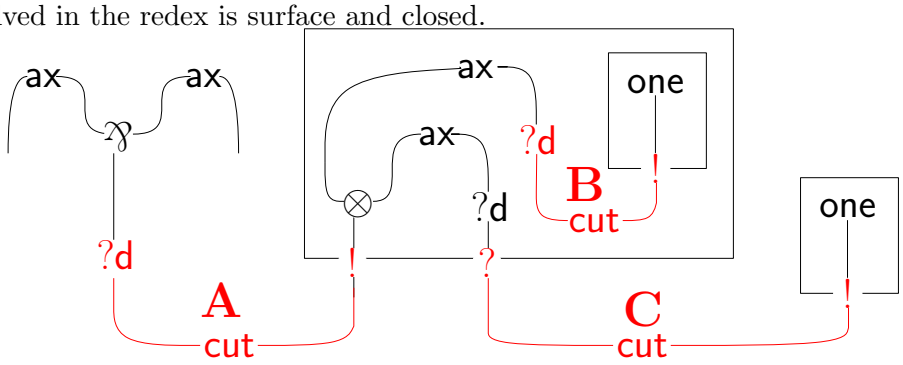


Figure 4.13: Example of a SMEYLL Proof Structure with Some Redexes

Since we have a structure for recursion, namely Y-boxes, the reduction of proof structures do not always terminate. Hence it makes sense to have a predicate for convergence:

Definition 4.7 (Convergence of SMEYLL Proof Structure Reduction)**.** We write $R{\Downarrow}$ if there exists a SMEYLL proof structure S satisfying $R \rightsquigarrow^* S$ and $S \not\rightsquigarrow$.

We are going to define a subset of proof structures, namely *proof nets*, and mainly focus on that subset as in linear logic. However, a strong form of confluence can be already proved for proof structures:

Proposition 4.8 (Diamond Property)**.** *Let R be a SMEYLL proof structure. If $R \rightsquigarrow R_1$ and $R \rightsquigarrow R_2$, then either $R_1 = R_2$ or there exists a SMEYLL proof structure S satisfying $R_1 \rightsquigarrow S$ and $R_2 \rightsquigarrow S$.*

Proof. By case analysis. □

Corollary 4.9 (Confluence, Uniqueness of Normal Form)**.** *The reduction relation $\rightsquigarrow$ is confluent. Moreover, the normal form is unique for any SMEYLL proof structure.*

Proof. By the tiling argument as usual in abstract reduction systems. □

In the rest of the thesis, we also make use of another subset of the proof structures, namely *simple* structures. Being simple is often assumed in propositions. Note that the following definition is about the conclusions of a *structure*; a simple structure may contain any complex substructure, e.g. Y-boxes.

Definition 4.10 (Simple Structure)**.** A SMEYLL proof structure R is *simple* if the conclusions of R consist only of the symbols $\{1, \otimes, ⅋\}$.

Example 4.11. The proof structure in Figure 4.14 below (exactly the same as Figure 4.10) is a simple proof structure; as noted above, it contains non-simple substructures.

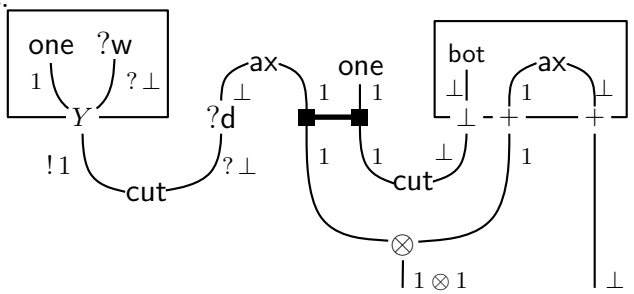


Figure 4.14: Example of SMEYLL Proof Structure (Shown Again)

Example 4.12. A sequence of reduction from the structure above is shown in Figure 4.15. Note that the reduction on the sync node cannot take place until two 1 nodes are immediately above the sync node.

4.2.4 SMEYLL Proof Nets

The SMEYLL proof structures include badly behaving ones, in a different meaning from the MELL case in which the proof structures considered as malicious are those do not have any sequent calculus proof as their counterpart. In our case the explanation does not stay valid, since the three nodes we introduced have not found their logical counterparts yet. Instead, we set our "correctness" as two properties: normalization of reduction and deadlock-freedom of the token machine we define later.

Remark 4.13. The relation between correctness of proof nets and deadlock-freedom has already been studied in [9], but the direction is different from ours. In short, they define a correctness criterion *by* deadlock-freedom while we define a correctness criterion *for* deadlock-freedom. In [9], correctness is given by a certain kind of deadlock-freedom by viewing proof structures as distributed systems, and they do use a notion of synchronization. However, their notion of synchronization is just to establish a path, and the tokens disappear immediately after synchronization. Moreover, synchronization of three or more tokens as in the quantum case is not considered nor straightforward to accommodate. Hence, for our purpose of introducing multi-token machines their viewpoint is not a satisfactory one.

The *correctness criterion* is defined as follows, using the notion of switching path as in linear logic, suitably accommodating sync nodes we introduced.

Definition 4.14 (Switching Path)**.** Given a SMEYLL proof structure R, a *switching path* is an undirected path on R (i.e. a path allowed to traverse a directed edge forward or backward) satisfying that

- on each $⅋$ node, the path uses at most one of the premises;
- on each $?c$ node, the path uses at most one of the premises;
- on each sync node, the path uses at most one of the conclusions;

and the path does not use any edge twice or more.

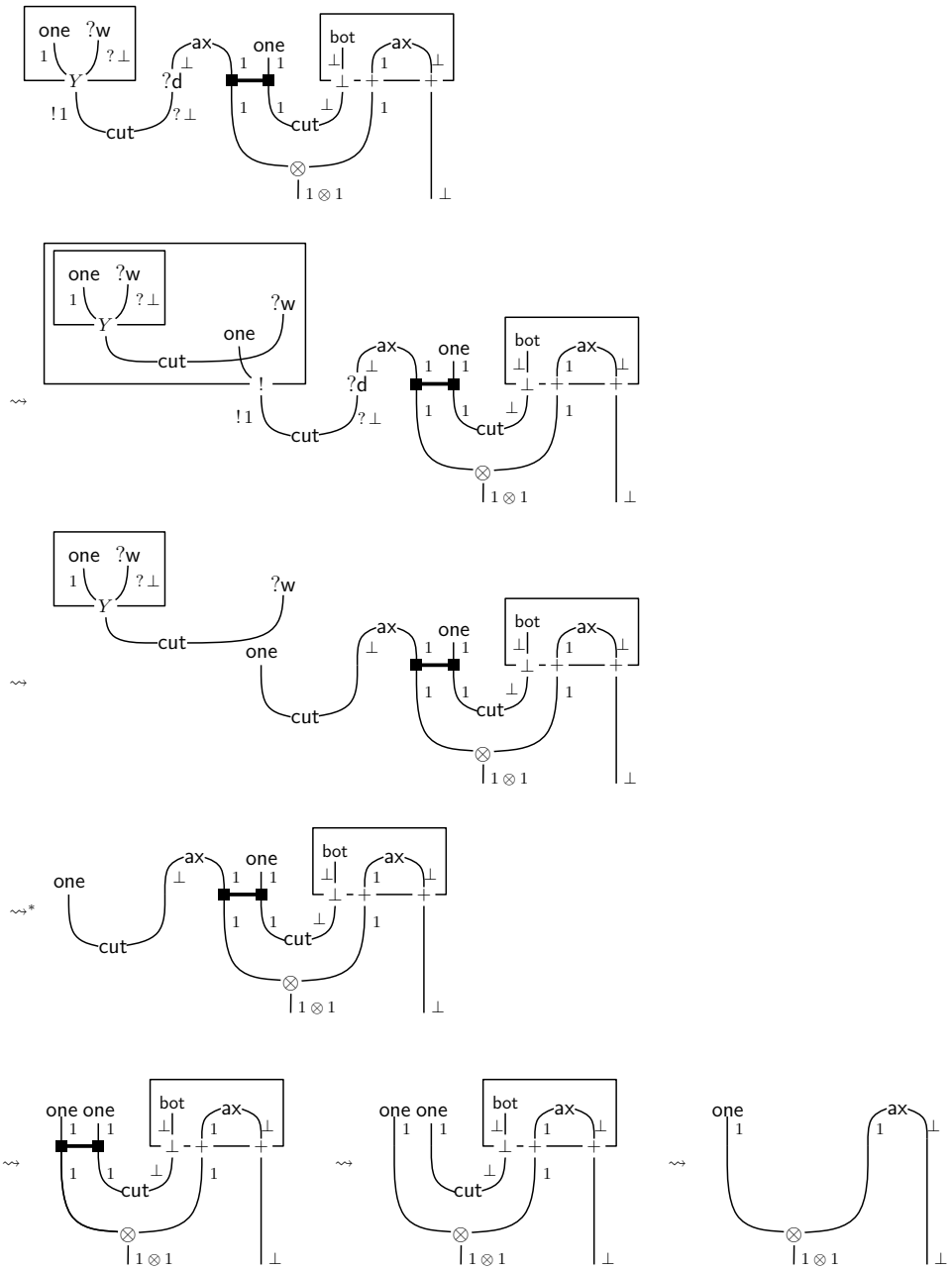


Figure 4.15: Example of Reductions

Definition 4.15 (Correctness Criterion)**.** The *correctness criterion for SMEYLL proof nets* states that given a SMEYLL proof structure R, all of its switching paths are acyclic. Such a path is said to be *correct*.

Definition 4.16 (SMEYLL Proof Net)**.** A *SMEYLL proof net* is a SMEYLL proof structure of which surface structure satisfies the correctness criterion and the content of every box in it is itself a SMEYLL proof net.

Example 4.17. The proof structure in Figure 4.14 indeed satisfies the correctness criterion, and is thus a SMEYLL proof net. It is clear that we have to use both of the conclusions of the unique sync node in order to form a cyclic path, but such a path cannot be a switching path by definition.

An example of SMEYLL proof structure that does not satisfy the correctness criterion is shown in Figure 4.16: there is a cyclic switching path shown as a red dashed path in the figure. Observe that the path does use at most one conclusion

per one sync node.

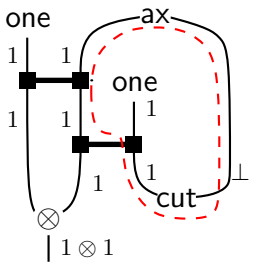


Figure 4.16: Example of Incorrect SMEYLL Proof Structure

As in linear logic, the correctness is preserved by reduction. Moreover, simple SMEYLL proof nets satisfies progress property, and cut elimination property as a corollary. Here the proof of progress property is fairly non-trivial, mainly because of $\perp$-boxes: the fact that *any* net may be contained in a $\perp$-box and may come out to the surface by opening a $\perp$-box prohibits naive induction.

Proposition 4.18. *Let R be a SMEYLL proof net and $R \rightsquigarrow S$. Then S also satisfies the correctness criterion, hence it is also a SMEYLL proof net.*

Proof. By case analysis. Note that the correctness criterion requires to use at most one *conclusion* on each sync node, so $\rightsquigarrow_{\mathrm{s}}$ rule cannot introduce a cycle if R satisfies the correctness criterion. □

Theorem 4.19 (Progress of SMEYLL Net)**.** *Let R be a simple SMEYLL proof net. If R contains a cut or a sync node, then there exists S that satisfies $R \rightsquigarrow S$.*

To show Theorem 4.19, we need some auxiliary notions. First we define a subset of nodes in a net and a notion of normal forms w.r.t. sync and multiplicative reductions:

Definition 4.20. Given a SMEYLL proof net R, the set $\mathcal{O}_R$ is defined as {sync nodes, boxes (both exponential and $\perp$-box), and axioms at surface in R}.

Definition 4.21. A SMEYLL proof net is said to be an *SM-normal form* if no reduction can be applied on any sync node nor on multiplicative node (i.e. axiom, cut, $\otimes$ or $\parr$).

Then the following lemmas immediately hold.

Lemma 4.22 (Sync Normal Forms)**.** *If no reduction can be applied on any sync node, then the only nodes which can be above a sync node are: sync, $\perp$-box, axiom, or* 1 *node.*

Proof. By typing of the nodes, $\parr$, $c?$, $d?$, $w?$, !-box, and Y-box cannot be connected to a sync node. Note that both $!\,A$ and $?\,A$ are not positive. If a $\otimes$ node is immediately above a sync node, the sync node can be reduced. The rest are the four kinds of nodes as stated. □

Lemma 4.23. *Each edge of type $!\,A$ is a conclusion of a box (either an exponential box or a $\perp$-box).*

Proof. The conclusion(s) of the other nodes cannot have type $!\,A$. □

What we mainly make use of is the following kind of paths over nets. The notion of non-bouncing path is the same one (except for sync nodes) as in linear logic proof net, while the last three conditions are specific to our setting.

Definition 4.24 (Priority Path)**.** Given a SMEYLL proof net R, a *priority path* on R is a *non-bouncing path*, i.e. a directed path on the net R regarded as an undirected graph, starting from a node in the set $\mathcal{O}_R$ and satisfying the following:

- if the path enters an edge of a cut or an axiom node, it exits from the other edge,
- if the path enters an auxiliary conclusion of a box, it exits from the principal conclusion, and vice versa,
- if the path enters a premise of a $\otimes$, $\parr$, or sync node, it exits from a conclusion, and vice versa,

and additionally satisfying the following:

- the path only enters a sync node from a conclusion and exits from a premise,
- the path only enters a $\bot$-box from an auxiliary conclusion and exits from the principal conclusion $\bot$,
- the path only enters an axiom from the conclusion of type $\mathsf{1}$ and exits from the conclusion of type $\bot$,
- the path only enters an exponential box from the principal conclusion and exits from an auxiliary conclusion.

Lemma 4.25. *Let R be a SMEYLL proof net in SM-normal form, and r be a priority path. The following hold:*

1. *When going downwards, the path r can only visit $\parr$, $\otimes$, $?c$, or $?d$ nodes, entering from a premise and exiting from the conclusion. Moreover, when going downwards, no edge occurring in the path r is of type $!A$,*
2. *When going upwards, the path r can only visit the following nodes, in the following way.*
   - *sync nodes: r enters from a conclusion and exits from a premise.*
   - *$\bot$ nodes: r enters from an auxiliary conclusion whose type is not in the form $?A$, and exits from the principal conclusion.*
   - *$!$ and Y nodes: r enters from the principal conclusion and exits from an auxiliary conclusion.*
   - *$\mathsf{1}$ nodes: r enters from the conclusion.*
   - *axiom nodes: r enters from the conclusion of type $\mathsf{1}$ and exits from the one of type $\bot$.*

   Moreover, when going upwards, no edge occurring in r has type $?A$.

Proof. Since r starts from a node in $\mathcal{O}_R$, and since r is a priority path, the path r must start as stated in item 1. or item 2. We prove item 1. and 2. each by induction on the length n of the path r.

1. - If $n = 1$, the path r starts from an axiom, a !-box, or Y-box and ends satisfying all the requirements.

- If $n > 1$, let $r = r'e$ where e is an edge traversed downwards, and $r'e$ is a path that is the prefix of r of length $n-1$. By induction hypothesis r' satisfies item 1. and 2. The last node $\mathrm{snd}(e)$ can only be one of $⅋$, $\otimes$, $?c$, $?d$, cut nodes, or a conclusion of the net, simply because of the direction of the path.

  Moreover, the type of the last edge e cannot be in the form $!\,A$ since it implies that the path r' exits from the principal conclusion of an exponential box or from one of auxiliary conclusions of a $\bot$-box (by Lemma 4.23), which contradicts to the fact that r is a priority path.

2. - If $n = 1$, the path r starts from a sync node and ends satisfying all the requirements.
   - If $n > 1$, let $r = r'(a, a')$ where r' is the prefix of r of length $n-1$ and (a, a') is an edge traversed upwards. By induction hypothesis r' satisfies item 2. Because of direction, the last node a' can be one of: axiom, sync, !-box (from the principal conclusion), Y-box (from the principal conclusion), $\bot$-box (from one of auxiliary conclusions), $⅋$, $\otimes$, $?c$, $?d$, or $?w$. However, the last five kinds of nodes cannot be there by the following reason. If a' is $?c$, $?d$, or $?w$, then a must be one of $⅋$, $\otimes$, $?c$, $?d$ because of typing, which contradicts to the induction hypothesis, or a cut node connected to a principal conclusion of an exponential box or an auxiliary conclusion of a $\bot$-box, which contradicts to the fact that r is a priority path. If a' is $⅋$ or $\otimes$, then a must be a sync node since r' satisfies the induction hypothesis, which contradicts to the net R being in SM-normal form, or a cut node connected to either the dual of a' or an auxiliary conclusion of a $\bot$-box, the former contradicting to R being in SM-normal form and the latter contradicting to r being a priority path. Therefore, the node a' must be one of axiom, sync, !-box, Y-box, or $\bot$-box.

     Moreover, the last edge e cannot be in the form $?\,A$ by the same reason as why the node a' cannot be $?c$, $?d$, or $?w$.

□

Lemma 4.26. *Let R be a SMEYLL net in SM-normal form. Every priority path on R is a switching path.*

Proof. First observe that if a priority path contains at most one premise of each $⅋$ node, at most one premise of each $?c$ node, and at most one conclusion of each sync node, then it is a switching path.

Suppose that a priority path r uses both premises of a $⅋$ node a. Since R is finite, we can choose such a path of which every proper subpath does not use two premises of a $⅋$ node or a $?c$ node, nor more than one conclusion of a sync node. By Lemma 4.25, r must visit a node a downwards. Therefore r must be in the form $r = (a', a)r'$ as shown in Figure 4.17; the subpath r' (shown in red) is a cyclic switching path, which contradicts to the assumption that R is a proof net and thus satisfies the correctness criterion. The same argument applies for $?c$ nodes and sync nodes. □

Now we define the *priority order* that is the key notion to show Theorem 4.19.

Definition 4.27 (Priority Order)**.** Let R be a SMEYLL proof net in SM-normal form. The relation $\prec \subseteq \mathcal{O}_R \times \mathcal{O}_R$, called the *priority order* is given by $a \prec b$ if and only if there exists a priority path from the node a to the node b.

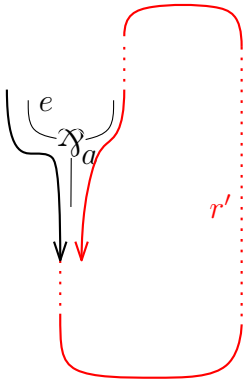


Figure 4.17: (Impossible) Priority Path

Proposition 4.28. *The priority order $\prec$ is a strict partial order.*

Proof. We prove that the relation is irreflexive and transitive.

- Irreflexivity: since a priority path is a switching path by Lemma 4.26, $a \prec a$ means there exists a cyclic switching path, contradicting to R being a proof net.
- Transitivity: assume $a \prec b$ and $b \prec c$. By definition there exist a priority path from a to b and one from b to c. Let $r = v_1....v_n$ be the priority path from $a = v_1$ to $b = v_n$ and $r' = w_1...w_m$ be the priority path from $b = w_1$ to $c = w_m$. We claim that b is the only node that the two paths have in common, and we can hence concatenate them and obtain a priority path from a to c. Assume that $l = w_j = k_i$ is the first node belonging to r' which belongs also to r. We follow r' from b to l and r from l to b. Let us call this path p, and check that it is non bouncing on l. Therefore p is a priority path, in contradiction with the fact that $b \prec b$ does not hold. We observe that p enters l as r' and exits l as r. The node l cannot be a cut, otherwise l would not be the first node which belongs to both paths. For all the other cases, Lemma 4.25 guarantees that, if l is a node of type $\otimes, \parr, ?c, ?d$, then r' enters from a premise, and r exits from a conclusion. The exact opposite is true if l is a sync node. If l is a $\bot$-box r' enters from an auxiliary conclusion, r exits from the principal conclusion. The opposite is true in case l is an exponential box. If l is an axiom, it is polarized; r' enters from a the positive conclusion, while r exits from the negative conclusion.

□

We can state the following lemma about the priority order.

Lemma 4.29. *Any exponential box maximal with respect to the priority order $\prec$ is closed.*

Proof. Let b be an exponential box maximal with respect to $\prec$. Each auxiliary conclusion $?A$ needs to be a hereditary premise of a cut node c. The path r descending from $?A$ to the cut node c is a priority path; the extension of r with the other premise C of c is still a priority path that is now ascending. By Lemma 4.25, the source of C could be either a 1 node, which is not possible because of the type, or a node in $\mathcal{O}_R$, which is in turn against maximality of b. Therefore, b cannot have any auxiliary conclusion. □

Lemma 4.30. *Let R be a SMEYLL proof net in SM normal form. $\mathcal{O}_R$ is empty if and only if there are no cuts in the net R.*

Proof. Assume $\mathcal{O}_R$ is empty, then R is an MLL net (with $\mathsf{1}$ nodes also); if there is a cut, we could perform a multiplicative reduction. Assume $\mathcal{O}_R$ is not empty. If there is a box or a polarized axiom, its principal conclusion needs to be cut, because it does not appear in the conclusions. If there are sync nodes, but no boxes or axioms, we could apply an s reduction. □

Finally, we can prove Theorem 4.19.

Proof. (of Theorem 4.19)

Let R be as in Theorem 4.19. If R is not in SM normal form, a sync or multiplicative reduction is possible by definition. If R is in SM normal form, and contains cuts, by Lemma 4.30, $\mathcal{O}_R$ is non-empty. We find a valid reduction step by case analysis.

- If $\mathcal{O}_R$ contains a *maximal* node l that is not an exponential box, we focus on it.
  - *l is a sync node.* Any path moving upward from l is a priority path. By using lemma 4.22 and the fact that l is maximal in $\mathcal{O}_R$, we know that above l there can only be $\mathsf{1}$ nodes. An *s.el* reduction hence applies.
  - *l is a $\bot$-box or a polarized axiom.* Let $\bot$ be the principal conclusion of the box or the negative conclusion of the axiom. Since it cannot appear in the conclusions, $\bot$ must be a hereditary premise of a cut c. We find c by descending from $\bot$. Since the path descending from l to the node c is a priority path, by Lemma 4.25 we know that the first node entered by r after the cut can only be a $\mathsf{1}$ node, because any other possibility would belong to $\mathcal{O}_R$ and is excluded by the maximality of l. By typing, this implies that the node c is in fact immediately below l. Hence a reduction applies (either bot.el or axiom-cut).
- Otherwise, we choose a node l as follows.
  - If $\mathcal{O}_R$ contains only exponential boxes, we observe that all cuts have premises of type $?A, !A$, and the $!A$ premise is principal conclusion of an exponential box. Let l be such a box.
  - If $\mathcal{O}_R$ contains nodes which are not exponential boxes, let l be any such node.

  If l is already a maximal exponential box, let $b_{max} := l$, otherwise we choose a maximal exponential box b_{max} such that $l \prec b_{max}$. The key properties that this careful construction guarantees is that for each exponential box b in the priority path from l to b_{max}:

  i. the principal conclusion $!A$ of b is premise of a cut c;

  ii. the other premise $?A^{\bot}$ of c *is not auxiliary conclusion* of a $\bot$-box.

  (i.) is true for l by construction; moreover, for every exponential box b which is reached by a priority path r, r can enter b only from the principal door, ascending from a cut (see case (1.c) in the proof of Lemma 4.25). (ii.) is true for any cut c which is reached by a priority path r, because if the cut has premises $!A, ?A^{\bot}$, r can only use the edge $?A^{\bot}$ to *descend* in c, and

must do so from a node which cannot be a $\bot$-box (because by Lemma 4.25, r exits a $\bot$-box only from the $\bot$ conclusion). We are now able to conclude.

By Lemma 4.29, b_{max} is a closed box. Because of (i.), the principal conclusion $!A$ of b_{max} is premise of a cut c. The other premise of c has type $?A^{\bot}$, and because of (ii.), $?A^{\bot}$ can only be conclusion of a node of type $?d, ?c, ?w$, or auxiliary conclusion of an exponential box. In each case a closed reduction applies.

□

Corollary 4.31 (Cut Elimination)**.** *Let R be a simple SMEYLL proof net. If $R \rightsquigarrow^* S$ and S is normal, then S does not contain a cut nor a sync node.*

Proof. Take the contrapositive of Theorem 4.19. □

Corollary 4.32. *Let R be a simple SMEYLL proof net. If R is normal, then R is an MLL proof net, i.e. it only consists of axiom, 1, $\otimes$, and $\parr$ nodes.*

Proof. By Corollary 4.31, R cannot contain a cut nor a sync node. If R contains a node whose conclusion is of type $?A$, it must be a hereditary premise of a cut since R is simple, thus contradicts to the fact that R does not have a cut. Similarly R cannot have a node with conclusion $\bot$ or $!A$. Thus R can only contain axiom, 1, $\otimes$, and $\parr$; the other kinds of nodes necessarily have one of the types mentioned above. □

4.3 Synchronous Interaction Abstract Machine

The section introduces the main contribution of the thesis, namely the *Synchronous Interaction Abstract Machine (SIAM for short)* on the SMEYLL proof nets. Like the Interaction Abstract Machine [23] (see Section 3.4), it is defined to be a transition system induced from transition rules on individual tokens. Unlike the IAM, a state of the system is no longer defined to be a position of *one* token, but a set of positions of *multiple* tokens. The difference makes the behaviors of the machine much more non-trivial, even though each transition rule is more or less the same as the IAM ones.

4.3.1 States

Definition 4.33 (Stack)**.** An *exponential signature* σ and a *stack* s are defined by the following BNF:

$$\sigma ::= * \;\mid\; l(\sigma) \;\mid\; r(\sigma) \;\mid\; \lceil \sigma, \sigma \rceil \;\mid\; y(\sigma, \sigma) \quad ,$$

$$\mathrm{s} ::= \epsilon \;\mid\; \delta.\mathrm{s} \;\mid\; l.\mathrm{s} \;\mid\; r.\mathrm{s} \;\mid\; \sigma.\mathrm{s} \quad ,$$

where ϵ denotes an empty stack and the dot (.) denotes concatenation. Concatenation s.ϵ or ϵ.s with an empty stack is defined to be s.

We have two non-standard symbols in the definition above, namely $y(-,-)$ and δ: the former handles recursion, while the latter is used to mark a new kind of tokens (*dereliction tokens*) appearing later in the section. A stack specify an occurrence of an atom or a modality in a formula:

Definition 4.34 (Occurrence Indication)**.** Let A be a SMEYLL logical formula and s be a stack. The stack s *indicates an occurrence of an atom* α (resp. *of a modality* $\spadesuit$) *in* A if $\mathrm{s}[A] = \alpha$ (resp. $\mathrm{s}[A] = \spadesuit$) holds, where $\mathrm{s}[A]$ is inductively defined by

- $\epsilon[\alpha] = \alpha$,
- $l.\mathrm{s}[A \diamond B] = \mathrm{s}[A]$,
- $r.\mathrm{s}[A \diamond B] = \mathrm{s}[B]$,
- $\sigma.\delta[\spadesuit A] = \spadesuit$,
- $\sigma.\mathrm{s}[\spadesuit A] = \mathrm{s}[A]$ if $\mathrm{s} \neq \delta$,
- $\mathrm{s}[A]$ is undefined otherwise,

where $\alpha \in \{1, \bot\}$, $\diamond \in \{\otimes, \parr\}$, and $\spadesuit \in \{!, ?\}$.

Example 4.35. Let $A = !_1(\bot \otimes !_2\, 1)$, where the subscripts of !'s are just for distinction of the two occurrence (i.e. the subscripts are not in the syntax of logical formulas.) As defined above, the occurrence of $!_1$ is indicated by $*.\delta$ since $*.\delta[!_1(\bot \otimes !_2\, 1)] = !_1$, while the occurrence of $!_2$ is indicated by a stack $*.r.*.\delta$, calculated as $*.r.*.\delta[!_1(\bot \otimes !_2\, 1)] = r.*.\delta[\bot \otimes !_2\, 1] = *.\delta[!_2\, 1] = !_2$. The occurrences of units $\bot$ and 1 in the formula A are indicated by $*.l$ and $*.r.*$, respectively.

Next we define *positions*. A position expresses where and with what status a token is running over a net.

Definition 4.36. Given a SMEYLL proof net R, a *position* in the net R is a triple $(e, \mathrm{s}, \mathrm{t})$ where e is an edge (including those inside boxes) of R and s, t are stacks respectively called a *formula stack* and a *box stack*, where s is either δ or a stack indicating an occurrence of an atom or a modality in the type A of the edge e and t is a stack consisting only of exponential signatures. We write $\mathbf{p}, \mathbf{q}$ for metavariables of positions.

The *direction* of a position $\mathbf{p} = (e, \mathrm{s}, \mathrm{t})$, denoted by $\mathtt{dir}(\mathbf{p})$, is an element of the set $\{\uparrow, \downarrow, \leftrightarrow\}$ defined by

- $\mathtt{dir}(\mathbf{p}) = \uparrow$ if s indicates an occurrence of a ! modality or a negative atom $\bot$,
- $\mathtt{dir}(\mathbf{p}) = \downarrow$ if s indicates an occurrence of a ? modality or a positive atom 1,
- $\mathtt{dir}(\mathbf{p}) = \leftrightarrow$ if $\mathrm{s} = \delta$ or e is the conclusion of a $\bot$ node.

The set of all positions in a net R is denoted by Pos_R. Several subsets of Pos_R will be used later:

- Init_R is the set of all positions $(e, \mathrm{s}, \mathrm{t})$ where e is a conclusion of R, $\mathtt{dir}(\mathbf{p}) = \uparrow$, and moreover all the exponential signatures in t are $*$.
- Fin_R is the set of all positions $(e, \mathrm{s}, \mathrm{t})$ where e is a conclusion of R and $\mathtt{dir}(\mathbf{p}) = \downarrow$.
- Ones_R is the set of all positions $(e, \mathrm{s}, \mathrm{t})$ where e is the conclusion of a 1 node in R.
- Der_R is the set of all positions $(e, \mathrm{s}, \mathrm{t})$ where e is the conclusion of a dereliction node in R.
- Start_R is defined to be $\mathrm{Init}_R \cup \mathrm{Ones}_R \cup \mathrm{Der}_R$.

- Stable_R is defined to be the set of all positions $\mathbf{p}$ with $\mathtt{dir}(\mathbf{p}) = \leftrightarrow$.

The requirement that the exponential signatures in t are $*$ in the definition of Init_R is for simplicity of presentation: the discussion later in the section can be done if we do not have the requirement, but the relaxation of the definition only yields redundancy and makes no essential benefit.

Definition 4.37 (SIAM State and Token)**.** Given a SMEYLL proof net R, a *state* $\mathrm{T} = (\mathrm{T}, \mathrm{orig}_{\mathrm{T}})$ over R consists of a set T of tokens and a function[3] $\mathrm{orig}_{\mathrm{T}} \colon \mathrm{T} \to \mathrm{Start}_R$. When $(e, \mathrm{s}, \mathrm{t}) \in \mathrm{T}$, we also say that "a *token* with the stacks (s, t) is on the edge e". The set of all states on R is denoted by $\mathcal{S}_R$.

In the definition of states, the function $\mathrm{orig}_{\mathrm{T}}$ is meant to map each position in T to its original position.

Definition 4.38 (Initial State)**.** Given a SMEYLL proof net R, the *initial state* is the state $\mathbf{I}_R = (\mathrm{Init}_R, \mathrm{orig}_{\mathbf{I}_R})$ where $\mathrm{orig}_{\mathbf{I}_R}$ is the inclusion function $\mathrm{Init}_R \hookrightarrow \mathrm{Start}_R$.

Definition 4.39 (Final State)**.** Given a SMEYLL proof net R, a *final state* is a state $(\mathrm{T}, \mathrm{orig}_{\mathrm{T}})$ where $\mathrm{T} \subseteq \mathrm{Fin}_R \cup \mathrm{Stable}_R$.

4.3.2 Transition Rules

So far we defined notions to describe "snapshots" of moving tokens. In the section we define the *transition rules* and explain their intuition. While many of them stay local and on a single token, we have several transition rules that involve multiple tokens at the same time. To define the transition rules we need to define sets of box stacks:

Definition 4.40. Let R be a SMEYLL proof net and $\mathrm{T} = (\mathrm{T}, \mathrm{orig}_{\mathrm{T}})$ be a state over R. For each substructure $S \in \{R\} \cup \{R' \mid R' \text{ is a content of a box in R}\}$, the set $\mathrm{Copy}_{\mathrm{T}}(S)$ is defined as follows.

$$\mathrm{Copy}_{\mathrm{T}}(S) = \begin{cases} \{\epsilon\} & \text{if } S = R \\ \{\mathrm{t} \mid (e, \mathrm{s}, \mathrm{t}) \in \mathrm{T} \text{ and } e \text{ is the principal conclusion of } S\} \end{cases}$$

Definition 4.41 (Transition Rules)**.** The *transition relation* of the SIAM is a binary relation $\to \subset \mathcal{S} \times \mathcal{S}$ induced by the *transition rules* pictorially defined in Figure 4.18 and 4.19, with the following conventions:

- a token $(e, \mathrm{s}, \mathrm{t})$ with the stacks (s, t) on the edge e is depicted by a bullet $(\bullet)$ accompanied by the pair of stacks (s, t) drawn on (or next to) the edge e in a picture.

- each transition rule except those marked with $(i), (ii), (iii)$ *moves* one token. If $\mathbf{p} = (e, \mathrm{s}, \mathrm{t})$ is drawn on the left-hand side of a picture and $\mathbf{p}' = (e', \mathrm{s}', \mathrm{t}')$ is drawn on the right-hand side, then the relation $\to$ is defined to be $(\{\mathbf{p}\} \cup \mathrm{T}, \mathrm{orig}_{(\{\mathbf{p}\} \cup \mathrm{T})}) \to (\{\mathbf{p}'\} \cup \mathrm{T}, \mathrm{orig}_{(\{\mathbf{p}'\} \cup \mathrm{T})})$ for all T, where $\mathrm{orig}_{(\{\mathbf{p}'\} \cup \mathrm{T})}(\mathbf{p}') = \mathrm{orig}_{(\{\mathbf{p}\} \cup \mathrm{T})}(\mathbf{p})$ and $\mathrm{orig}_{(\{\mathbf{p}'\} \cup \mathrm{T})}(\mathbf{q}) = \mathrm{orig}_{(\{\mathbf{p}\} \cup \mathrm{T})}(\mathbf{q})$ if $\mathbf{q} \neq \mathbf{p}$.

[3] In [18], a state T explicitly bears the image of the function $\mathrm{orig}_{\mathrm{T}}$ instead of the function itself, while in [19] we use $\mathrm{orig}_{\mathrm{T}}$; the two representations are essentially equivalent, and make no difference in practice.

- the rules marked with (i) *generate* a token if the position newly-added to the state is not already contained in the image of $\mathrm{orig}_{\mathrm{T}}$ and also $\mathrm{t} \in \mathrm{Copy}_{\mathrm{T}}(S)$, where S is the substructure the edge belongs to. Explicitly, the rules relate $(\mathrm{T}, \mathrm{orig}_{\mathrm{T}}) \to (\{\mathbf{p}\} \cup \mathrm{T}, \mathrm{orig}_{(\{\mathbf{p}\} \cup \mathrm{T})})$ if $\mathbf{p} \notin \mathrm{Im}(\mathrm{orig}_{\mathrm{T}})$, where $\mathbf{p} = (e, \mathrm{s}, \mathrm{t})$ as depicted on the right-hand side and $\mathrm{orig}_{(\{\mathbf{p}\} \cup \mathrm{T})}(\mathbf{p}) = \mathbf{p}$. The token generated on the conclusion of a $?d$ node is called a *dereliction token*.

- the rule marked by (ii) also requires $\mathrm{t} \in \mathrm{Copy}_{\mathrm{T}}(S)$, where S is the content of the $\bot$-box.

- the rule marked by (iii) moves *multiple* tokens simultaneously. We have a requirement to apply the rule: every positive atom occurring in the premises of the sync node is indicated by a token, and all the involved tokens have the same box stack t.

Besides the conditions $(i), (ii), (iii)$, it is worth mentioning the rule for Y-box. When a token arrives at its principal conclusion, it first enters the Y-box and move onto the edge typed by A as if it is a !-box. After traveling inside the Y-box, the token may reach the principal door from the edge typed by $?A^{\bot}$. This behavior corresponds to a recursive call of the function expressed as the Y-box: to capture the situation, the token goes onto the edge typed by A, with one more y symbol in the formula stack. Intuitively, the number n of the y symbols expresses that the token is executing the n-th recursive call. When the execution of a recursive call finishes, the token now starts to return the result, ultimately reaching the edge typed by A with the direction $\downarrow$; then it pass the information to the previous call, with the number of y symbol decreased by 1. If the token does not contain y symbol on the top of the stack, it means that the entire execution has finished, and thus it exits the Y-box.

Example 4.42. A sequence of transitions on the SMEYLL net in Example 4.6 is shown in Figure 4.20. The token in the initial state is prohibited to enter the $\bot$-box until a token comes to the $\bot$ node; the token from the 1 node above the sync node is also blocked until another token arrives at the other premise of the sync node.

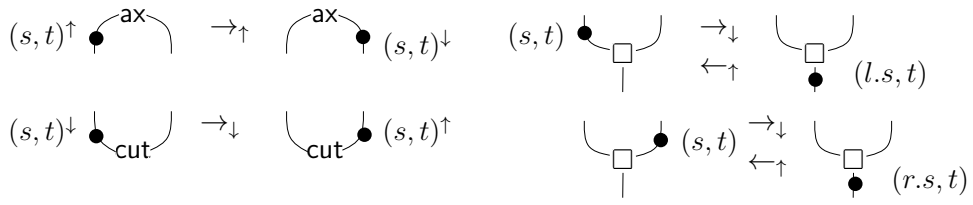
A fact to note here is that the definition of the transition rules of the SIAM is *parametric* on the box stack. Precisely,

Fact 4.43 (Parametricity)**.** Every transition rule is defined parametrically with respect to box stacks. That means, if a transition
$\{(e_1, \mathrm{s}_1, \mathrm{t}), (e_2, \mathrm{s}_2, \mathrm{t}), \ldots (e_n, \mathrm{s}_n, \mathrm{t})\} \cup \mathrm{T}$
$\to \{(e'_1, \mathrm{s}'_1, \mathrm{t}'_1), (e'_2, \mathrm{s}'_2, \mathrm{t}'_1), \ldots (e'_m, \mathrm{s}'_m, \mathrm{t}'_m)\} \cup \mathrm{T}$
is possible with box stacks t, then there exists a transition
$\{(e_1, \mathrm{s}_1, u), (e_2, \mathrm{s}_2, u), \ldots (e_n, \mathrm{s}_n, u)\} \cup T$
$\to \{(e'_1, \mathrm{s}'_1, u'_1), (e'_2, \mathrm{s}'_2, u'_2), \ldots (e'_m, \mathrm{s}'_m, u'_m)\} \cup T$
for any box stacks u. Here $n = m$ where n is the number of saturated tokens if the rule is sync; $n = m = 2$ with a token on an edge and another token in $\mathrm{Copy}_{\mathrm{T}}(S)$ if the rule is $\bot$-box; $n = 1$, $m = 2$ with a token in $\mathrm{Copy}_{\mathrm{T}}(S)$ and another token added to the state if the rule is token generation; otherwise $n = m = 1$.

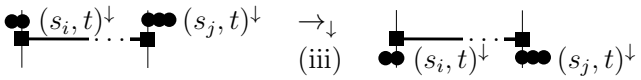
Having explained the rules, the *Synchronous Interaction Abstract Machine (SIAM) on net R* is defined as a transition system over the states of R.

Definition 4.44 (Synchronous Interaction Abstract Machine)**.** Let R be a SMEYLL proof net. The *Synchronous Interaction Abstract Machine (SIAM) on net R*, denoted by $\mathcal{M}_R$, is a transition system $(\mathcal{S}_R, \mathbf{I}_R, \to)$ with $\mathbf{I}_R$ the initial state.

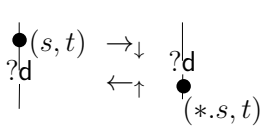
Multiplicatives

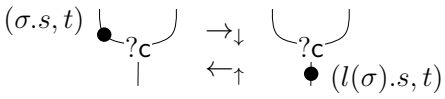


Sync



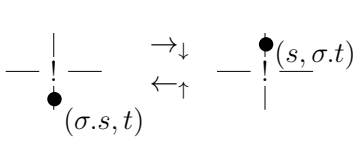
Exponential Nodes

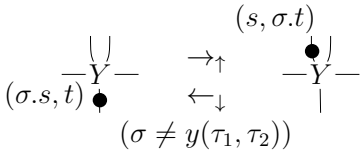




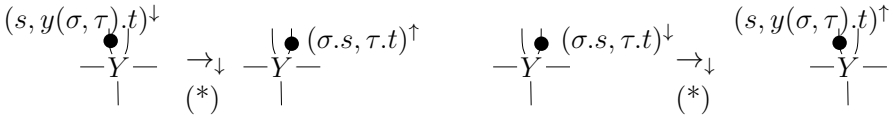
(and similarly for the right premiss)

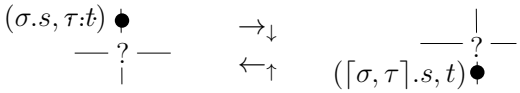
Exponential Boxes



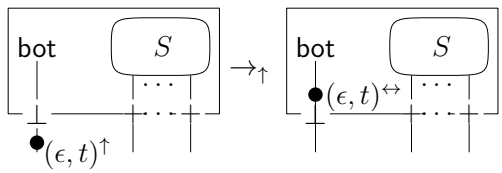


$(\sigma \neq y(\tau_1,\tau_2))$





$\perp$-boxes



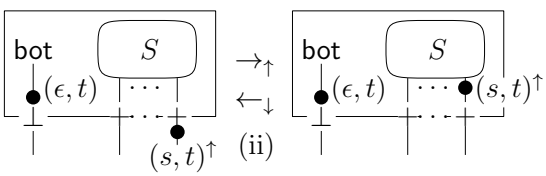


Figure 4.18: SIAM Transition Rules (i)

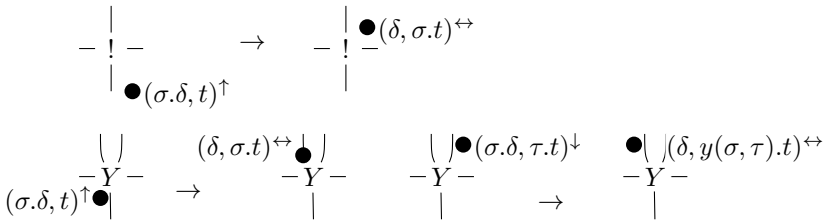


Figure 4.19: SIAM Transition Rules (ii): Exponential Transitions to Stable Positions

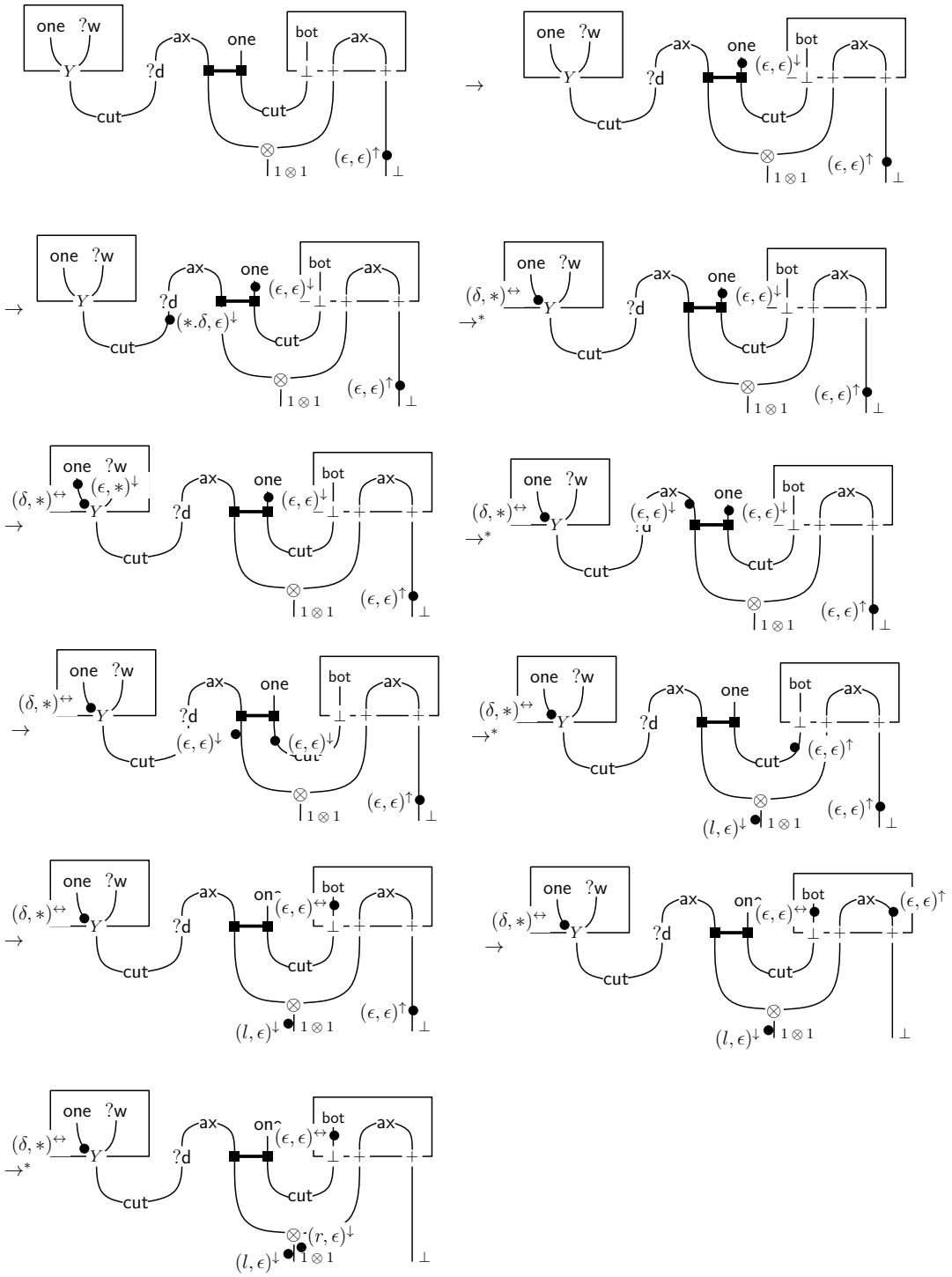


Figure 4.20: Example of Transitions

Definition 4.45. Let R be a SMEYLL proof net. An (either finite or infinite) maximal sequence of transitions $\mathbf{I}_R \to \mathrm{T}_1 \to \ldots \to \mathrm{T}_n \to$ is called a *run* of the SIAM $\mathcal{M}_R$. We write $\mathrm{T} \not\to$ if no transition rule can applied on the state T. A non-final state $\mathrm{T} \not\to$ is said to be *deadlocked*. If there exists a run $\mathbf{I}_R \to \mathrm{T}_1 \to \ldots \to \mathrm{T}_n \not\to$, the SIAM $\mathcal{M}_R$ is said to *terminate* or *converge*. Similarly, if there exists an infinite run $\mathbf{I}_R \to \mathrm{T}_1 \to \ldots \to \mathrm{T}_n \to \ldots$, the SIAM $\mathcal{M}_R$ is said to *diverge*; if there exists an run $\mathbf{I}_R \to \mathrm{T}_1 \to \ldots \to \mathrm{T}_n \not\to$ with T_n deadlocked, the SIAM $\mathcal{M}_R$ is said to *deadlock*.

We often simply say the *machine* to mention the SIAM when it is unambiguous. As an abstract reduction system, it is straightforward to check that the machine $\mathcal{M}_R$ on a net R enjoys some desirable properties:

Proposition 4.46 (Diamond Property)**.** *Given any SMEYLL net R, the SIAM $\mathcal{M}_R$ has the diamond property.*

Proof. By case analysis on each pair of rules: observe that a transition on a token cannot prohibit a transition on any of the other tokens. □

Corollary 4.47 (Confluence, Uniqueness of Normal Form)**.** *Given any SMEYLL net R, the SIAM $\mathcal{M}_R$ is confluent. Moreover, its final state reachable from the initial state is unique.*

Proof. By the tiling argument. (See Section 3.2.) □

Finally, the *token machine semantics* of a SMEYLL proof net is given as follows.

Definition 4.48 (Token Machine Semantics)**.** Let R be a SMEYLL proof net. The *token machine semantics* $[\![R]\!]$ *of* R is either a symbol Ω or a partial function $[\![R]\!] \colon \mathrm{Init}_R \rightharpoonup \mathrm{Fin}_R$ defined by

- $[\![R]\!] = \Omega$ if $\mathcal{M}_R$ diverges.
- $[\![R]\!](\mathbf{p}) = \mathbf{q}$ if $\mathcal{M}_R$ converges and $\mathrm{orig}(\mathbf{q}) = \mathbf{p}$ for $\mathbf{q} \in \mathrm{T}$ where T is the final state of $\mathcal{M}_R$.

Note that in the definition above, a machine $\mathcal{M}_R$ diverges if there *exists* one or more tokens that continue to move forever, even if several tokens can reach their final positions. This might seem just a matter of definition, but it makes an essential difference later.

4.3.3 Invariance

A crucial property that the SIAM should satisfy is *invariance*, meaning that the behavior of a machine is preserved by net reduction (and thus it).

Theorem 4.49 (Invariance)**.** *Let R be a SMEYLL proof net and $R \rightsquigarrow S$. Then $[\![R]\!] = [\![S]\!]$ (with the obvious identification of conclusions).*

The proof is more non-trivial than the MELL case because not only of being multi-token but also of existence of divergence. A technical difficulty is that the key lemmas to prove invariance in the MELL case do not naively extend to our SMEYLL case due to those reasons. To show the theorem, we first define an auxiliary notion of *transformation map* that "translates" the states in $\mathcal{M}_R$ into those in $\mathcal{M}_S$ (we avoid to name it *translation* because it becomes confusing later in the thesis).

Definition 4.50 (Transformation Map)**.** Given a reducible SMEYLL net R, a redex r in R, and a reduction on that redex $R \rightsquigarrow S$, the *transformation map from R to S* is a partial function $\mathrm{trsf}_{R,r,S} \colon \mathcal{S}_R \rightharpoonup \mathcal{S}_S$ given by the following way.

- Let the reduction be an ax reduction. Then $\mathrm{trsf}_{R,r,S}(e, \mathrm{s}, \mathrm{t})$ is defined to be $(e, \mathrm{s}, \mathrm{t})$ if the edge e is not in the redex; otherwise, let e_1, e_2, e_3 be the edges in the redex and e' be the edge in the reduct, as shown in Figure 4.21. Then,

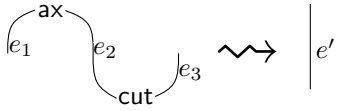


Figure 4.21: Edges in the ax Redex

  $\mathrm{trsf}_{R,r,S}(e_i, \mathrm{s}, \epsilon)$ is defined to be $(e', \mathrm{s}, \epsilon)$ for each $i \in \{1, 2, 3\}$. Since we require surface reduction, the box stack must be ϵ; otherwise $\mathrm{trsf}_{R,r,S}(e_i, \mathrm{s}, \epsilon)$ is undefined.

- The description above can be rigorously depicted as Figure 4.22: a position $(e, \mathrm{s}, \epsilon)$ is represented by putting a bullet $(\bullet)$ and the pair of stacks (s, ϵ) next to the edge e, and the mapping is represented by the dashed arrows. If the edge e of a position $(e, \mathrm{s}, \mathrm{t})$ is in the redex, but the stacks do not match with the stacks depicted next to the edge, then $\mathrm{trsf}_{R,r,S}(e, \mathrm{s}, \mathrm{t})$ is undefined. Positions not on the edges depicted are assumed to remain the same (i.e. the image by $\mathrm{trsf}_{R,r,S}(e, \mathrm{s}, \mathrm{t})$ is defined to be identical to the position $(e, \mathrm{s}, \mathrm{t})$). We define $\mathrm{trsf}_{R,r,S}$ for the other cases of reduction rules

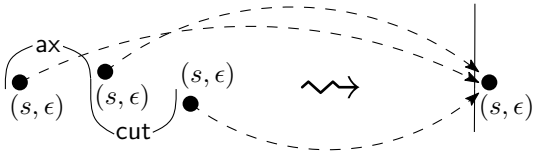


Figure 4.22: $\mathrm{trsf}_{R,r,S}$ on ax Reduction, Pictorially

  in Figure 4.23 following the convention described, plus a convention that a dashed arrow pointing to a crossing $(\times)$ means the map is undefined on the position.

We write trsf for $\mathrm{trsf}_{R,r,S}$ when no ambiguity occurs.

Remark 4.51. The definition of trsf for some reduction rules deserves detailed explanation.

- The mapping on the edges around the ⅋ node in the case of ⊗-⅋ rule is an analogue of those around ⊗. Tokens on the *left* premise are mapped to those on the right premise of the cut node on the *left*, and right to right; tokens on the conclusion are mapped to left or right according to the symbol at the top of the formula stack.

- In the definition of trsf for $?d$ reduction rule, the dereliction token coming from the $?d$ node in the redex is deleted. The tokens in the !-box are mapped to the content of the box that is now at surface, with one $*$ symbol lost from the bottom of the box stack.

- If the reduction rule is y, we distinguish two cases: if a token in the Y-box does not have the y symbol at the bottom of the box stack, it is mapped to the content not in the Y-box in the reduct. This is because having no y symbol at the bottom means it is the first function call of the recursive function (that is now unfolded as the content of the !-box in the reduct).

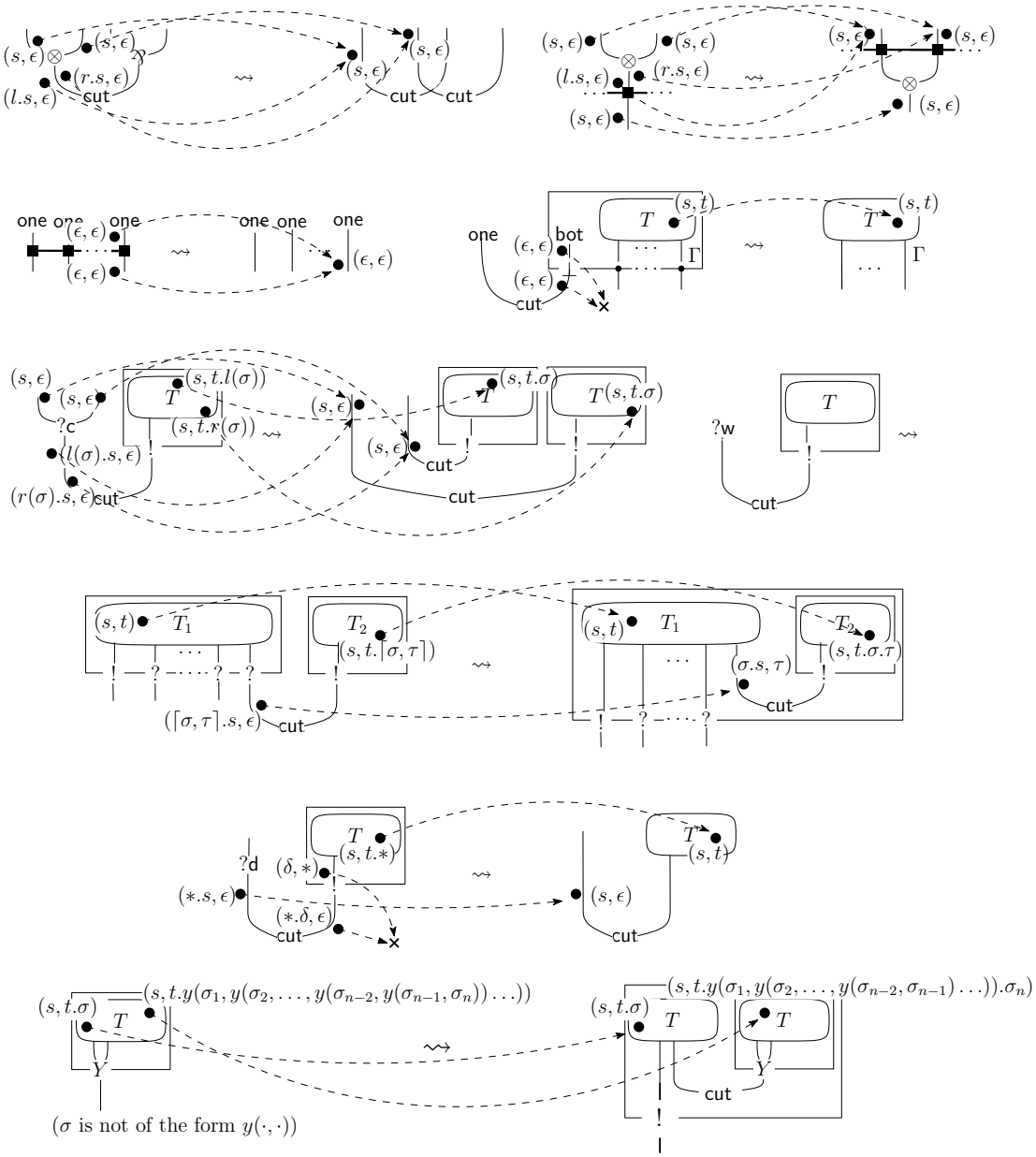


Figure 4.23: $\mathrm{trsf}_{R,r,S}$ on Other Reductions

If a token in the Y-box does have one or more y symbols at the bottom of the box stack, meaning it is the second or later call, then it is mapped to the content of the Y-box, losing one y symbol.

- No token can exist on the $?w$ redex

Intuitively, $\mathrm{trsf}_{R,r,S}$ sends a state T in $\mathcal{M}_R$ into another state U in $\mathcal{M}_S$ that is "equivalent" to T. The intuition is clarified by the following lemma.

Lemma 4.52 (Properties of Transformation Map)**.** *Let R be a reducible SMEYLL net, r be a redex in R, and $R \rightsquigarrow S$ be a reduction on that redex.*

1. *If* $\mathrm{T} \to \mathrm{U}$ *in* $\mathcal{M}_R$*, then* $\mathrm{trsf}(\mathrm{T}) \to^* \mathrm{trsf}(\mathrm{U})$ *in* $\mathcal{M}_S$*.*
2. *If* T *is initial, then* trsf(T) *is initial.*
3. *If* T *is final, then* trsf(T) *is final.*

4. *If an (either finite or infinite) sequence of transitions* $\mathbf{I} \to \mathrm{T} \to \mathrm{T}_1 \ldots \to \mathrm{T}_n \to \ldots$ *in* $\mathcal{M}_R$ *converges, diverges, or deadlocks, then* $\mathrm{trsf}(\mathbf{I}) \to^* \mathrm{trsf}(\mathrm{T}) \to^* \mathrm{trsf}(\mathrm{T}_1) \ldots \to^* \mathrm{trsf}(\mathrm{T}_n) \to^* \ldots$ *converges, diverges, or deadlocks, respectively.*

Lemma 4.53. *Let* $\mathbf{I}_R \to \ldots \to \mathrm{T} = (\mathrm{T}, \mathrm{orig}_\mathrm{T})$ *be a run in* $\mathcal{M}_R$ *and* $\mathbf{I}_S \to \ldots \to \mathrm{trsf}(\mathrm{T}) = (\mathrm{U}, \mathrm{orig}_\mathrm{U})$ *be the corresponding sequence of transitions in* $\mathcal{M}_S$*. For each* $\mathbf{p} \in \mathrm{T}$*, we have the following:*

- $\mathrm{orig}_\mathrm{T}(\mathbf{p}) \in \mathrm{Init}_R$ *iff* $\mathrm{orig}_\mathrm{U}(\mathrm{trsf}(\mathbf{p})) \in \mathrm{Init}_S = \mathrm{Init}_R$*.*
- *If* $\mathrm{orig}_\mathrm{T}(\mathbf{p}) \in \mathrm{Init}_R$ *then* $\mathrm{orig}_\mathrm{T}(\mathbf{p}) = \mathrm{orig}_\mathrm{U}(\mathrm{trsf}(\mathbf{p}))$*.*

Proof. By induction on the length of the run. If $\mathrm{T} = \mathbf{I}_R$ it is immediate by definition of trsf. If $\mathbf{I}_R = \mathrm{T}_0 \to \ldots \to \mathrm{T}_{n-1} \to \mathrm{T}_n$, it can be shown that tracing back from the positions in $\mathrm{trsf}(\mathrm{T}_n)$ reaches the same positions as $\mathrm{trsf}(\mathrm{T}_{n-1})$ by case analysis on reduction rules and transition rules. □

To show item 4. of Lemma 4.52, We define an auxiliary notion on transitions and prove an auxiliary lemma:

Definition 4.54. Given a SMEYLL proof net R and a reduction $R \rightsquigarrow S$, a transition $\mathrm{T} \to \mathrm{U}$ in the SIAM $\mathcal{M}_R$ is called a *collapsing transition* (resp. a *non-collapsing transition*) if the transition satisfies $\mathrm{trsf}(\mathrm{T}) = \mathrm{trsf}(\mathrm{U})$ (resp. $\mathrm{trsf}(\mathrm{T}) \to^+ \mathrm{trsf}(\mathrm{U})$).

Lemma 4.55. *Let* R *be a SMEYLL proof net and* $R \rightsquigarrow S$ *be a reduction. For any state* T *reachable from the initial state (i.e. a state satisfying* $\mathbf{I}_R \to^* \mathrm{T}$*), an infinite sequence of transitions* $\mathrm{T} \to \cdots$ *starting from* T *contains infinitely many non-collapsing transitions.*

Proof. Let us look at the case of ax-cut reduction: the other cases can be shown similarly. Let $R \rightsquigarrow_a R'$, $\mathrm{T} = (\mathrm{T}, \mathrm{orig}_\mathrm{T})$, and e_1, e_2, e_3 be the edges in Figure 4.24. Since T is finite, the set $\{(e_i, s, \epsilon) \mid i \in \{1,2,3\}\} \cap \mathrm{T}$ is also finite. Let n be

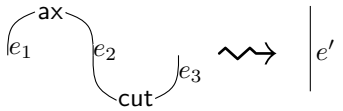


Figure 4.24: Edges of the Redex

the number of elements in this set of positions. The length of a sequence of transitions from T that only uses collapsing transitions is bounded by $2n$, since we cannot apply more than two collapsing transitions on each token. Thus an infinite sequence $\mathrm{T} \to \cdots$ of transitions starting from T must contain a non-collapsing transition. The argument above applies for every state in an infinite sequence of transition. Hence, any infinite sequence of transitions starting from the initial state contains infinitely many non-collapsing transitions. Similar argument can be done for the other rules. □

Proof. (of Lemma 4.52.)

1. First note that the statement 1. includes $\mathrm{trsf}(\mathrm{T}) = \mathrm{trsf}(\mathrm{U})$. If the transition $\mathrm{T} \to \mathrm{U}$ is not on the redex of $R \rightsquigarrow S$ then the claim holds, because the positions of tokens and the structure are the same except around the redex. Else we examine each case of reductions, where we have to consider only such a transition $\mathrm{T} \to \mathrm{U}$ that moves a token on the redex:

- $\underline{\rightarrow_a}$ The states T and U are mapped to $\mathrm{trsf}(\mathrm{T}) = \mathrm{trsf}(\mathrm{U})$ by definition of trsf.
- $\underline{\rightarrow_m}$ Similarly we verify that $\mathrm{trsf}(\mathrm{T}) = \mathrm{trsf}(\mathrm{U})$ (if the transition crosses $\otimes$ or $\parr$ node) or $\mathrm{trsf}(\mathrm{T}) \rightarrow \mathrm{trsf}(\mathrm{U})$ (if the transition crosses the cut node).
- $\underline{\rightarrow_s}$ If the transition crosses the $\otimes$ node then $\mathrm{trsf}(\mathrm{T}) = \mathrm{trsf}(\mathrm{U})$. If the transition crosses the sync node then $\mathrm{trsf}(\mathrm{T}) \rightarrow \mathrm{T}'_1 \rightarrow \ldots \rightarrow \mathrm{T}'_n \rightarrow \mathrm{trsf}(\mathrm{U})$, where the first transition crosses the sync node and each of the other transitions crosses the $\otimes$ node one by one.
- $\underline{\rightarrow_{s.el}}$ Similar to the case of $\rightarrow_a$.
- $\underline{\rightarrow_{bot.el}}$ If the transition crosses the cut node or enters the box then $\mathrm{trsf}(\mathrm{T}) = \mathrm{trsf}(\mathrm{U})$. Else, for the transition $\mathrm{T} \rightarrow \mathrm{U}$ in $\mathcal{M}_R$ inside the box allowed by a stable token, there is always a transition $\mathrm{trsf}(\mathrm{T}) \rightarrow \mathrm{trsf}(\mathrm{U})$ since the structure contained in R in the box is now at surface in S.
- $\underline{\rightarrow_c}$ If the transition crosses the $?c$ node then $\mathrm{trsf}(\mathrm{T}) = \mathrm{trsf}(\mathrm{U})$. Else $\mathrm{trsf}(\mathrm{T}) \rightarrow \mathrm{trsf}(\mathrm{U})$ since the transition rules are defined parametrically with respect to box stacks (thus if $\mathrm{T} \rightarrow \mathrm{U}$ is done with a box stack $\mathrm{t}.l(\sigma)$, $\mathrm{trsf}(\mathrm{T}) \rightarrow \mathrm{trsf}(\mathrm{U})$ can be done with box stack $\mathrm{t}.\sigma$.)
- For the other exponential rules the situation is similar: if the transition is on a dereliction token the states collapse, else $\mathrm{trsf}(\mathrm{T}) \rightarrow \mathrm{trsf}(\mathrm{U})$ is possible by the same rule as $\mathrm{T} \rightarrow \mathrm{U}$ with a different box stack.

2. By case analysis.
3. By case analysis again.
4. By Lemma 4.55, there are infinitely many transitions $\mathrm{T}_i \rightarrow \mathrm{T}_{i+1}$ that satisfies $\mathrm{trsf}(\mathrm{T}_i) \rightarrow \mathrm{trsf}(\mathrm{T}_{i+1})$. Therefore the sequence $\mathrm{trsf}(\mathbf{I}) \rightarrow^* \mathrm{trsf}(\mathrm{T}) \rightarrow^* \mathrm{trsf}(\mathrm{T}_1) \ldots \rightarrow^* \mathrm{trsf}(\mathrm{T}_n) \rightarrow^* \ldots$ is infinite.

□

Now we have set all the tools to show the invariance theorem.

Proof. (of Theorem 4.49).
First of all, since a reduction of a net does not changes the conclusions of the net, $\mathrm{Init}_R = \mathrm{Init}_S$ and $\mathrm{Fin}_R = \mathrm{Fin}_S$. Let us call them Init and Fin, respectively. By Lemma 4.52.4, if $\mathcal{M}_R$ diverges then $\mathcal{M}_S$ does so, too, which means that if $[\![R]\!] = \Omega$ then $[\![S]\!] = \Omega$. Conversely, if $\mathcal{M}_S$ diverges, then $\mathcal{M}_R$ must also diverge. This is because otherwise $\mathcal{M}_R$ has its unique final state T and its image under trsf is final; however then by diamond property of the SIAM $\mathcal{M}_S$ cannot diverge. Hence $[\![R]\!] = \Omega$ if and only if $[\![S]\!] = \Omega$.

Now suppose the machine $\mathcal{M}_R$ terminates in the final state $\mathrm{T} = (\mathrm{T}, \mathrm{orig}_\mathrm{T})$; by Lemma 4.52 and Lemma 4.47, $\mathcal{M}_S$ must terminate in its unique final state $\mathrm{U} = (\mathrm{U}, \mathrm{orig}_\mathrm{U})$. By Lemma 4.53 and Lemma 4.52, $\mathrm{trsf}(\mathrm{T}) = \mathrm{U}$ and $\mathrm{orig}_\mathrm{T}(\mathbf{p}) = \mathrm{orig}_\mathrm{U}(\mathrm{trsf}(\mathbf{p}))$ for a position $\mathbf{p} \in \mathrm{T}$ if $\mathrm{trsf}(\mathbf{p})$ is defined.

If $[\![R]\!](\mathbf{p}) = \mathbf{q}$ for some $\mathbf{p} \in \mathrm{Fin}$, by Lemma 4.53 $[\![S]\!](\mathrm{trsf}(\mathbf{p})) = \mathbf{q}$. Since all the positions in Fin must have ϵ, it is necessary that $\mathbf{p} = \mathrm{trsf}(\mathbf{p})$. Therefore $[\![S]\!](\mathbf{p}) = [\![S]\!](\mathrm{trsf}(\mathbf{p})) = \mathbf{q}$. Conversely, if $[\![S]\!](\mathbf{p}') = \mathbf{q}$, there must be a final position $\mathbf{p} \in \mathrm{T}$ satisfying $\mathrm{trsf}(\mathbf{p}) = \mathbf{p}'$ since T is final and by Lemma 4.52. By Lemma 4.53 $\mathrm{orig}(\mathbf{p}) \in \mathrm{Init}$, thus again by Lemma 4.53 $\mathrm{orig}(\mathbf{p}) = \mathrm{orig}(\mathbf{p}')$, hence $[\![R]\!](\mathbf{p}') = \mathrm{orig}(\mathbf{p}) = \mathbf{q}$.

□

4.3.4 Adequacy

Another important property we are going to show is *adequacy*, meaning the convergence of a proof net and that of its token machine coincide. In fact, this is true only for simple nets due to surface reduction and existence of $\bot$-box; however it is sufficient for our purpose, namely interpretation of closed terms. We prove the following:

Theorem 4.56 (Adequacy between SMEYLL Nets and SIAM)**.** *Let R be a simple SMEYLL net. Then $R\Downarrow$ if and only if $\mathcal{M}_R\Downarrow$.*

To show the theorem we will observe that termination of a net implies termination of the machine, and vice versa. This is achieved by looking at the *weight* of nets defined as follows.

Definition 4.57 (Weight)**.** Let R be a SMEYLL proof net, and assume that the SIAM $\mathcal{M}_R$ terminates in the final state T. The *weight* $W(\mathrm{T})$ of the state R is defined to be the sum of the number of dereliction tokens and the number of tokens started from 1 node, i.e. $W(\mathrm{T}) := \#\{\mathbf{p} \in \mathrm{T} \mid \mathrm{orig}_{\mathrm{T}} \in \mathrm{Der}_R \cup \mathrm{Ones}_R\}$.

Lemma 4.58. *Assume $R \to S$. We have that $W(\mathrm{T}) \geq W(\mathrm{trsf}(\mathrm{T}))$. Moreover, if $R \to S$ by the d-rule or $\bot$.el-rule, then $W(\mathrm{T}) > W(\mathrm{trsf}(\mathrm{T}))$.*

Proof. Can be proved by checking which tokens are "deleted" in the definition of trsf (Fig. 4.23). □

Lemma 4.59 (Mutual Termination)**.** *Let R be a simple SMEYLL net.*

1. *If a run (see Definition 4.45) of $\mathcal{M}_R$ terminates, then every sequence of reductions starting from R terminates;*

2. *If a sequence of reductions starting from R terminates, then every run of $\mathcal{M}_R$ terminates in a* final *state.*

Proof. 1. By assumption, there is a run of $\mathcal{M}_R$ which terminates in a state T. We define the weight of the net R as $W(R) := W(\mathrm{T})$. By Lemma 4.52, if $R \to S$, trsf maps the run of $\mathcal{M}_R$ into a run of $\mathcal{M}_S$ which terminates in the state $\mathrm{trsf}(\mathrm{T})$. By Lemma 4.58, $W(\mathrm{trsf}(\mathrm{T})) \leq W(\mathrm{T})$, hence $W(S) \leq W(R)$. Then it is not possible to have an infinite sequence of net reductions starting from R because: (i) each reduction that opens a box (d or *bot.el*) strictly decreases the weight of the net; (ii) there can be only a finite number of reductions that do not open any box.

2. By assumption, R reduces to a cut free net S, which has the form described in Corollary 4.32. On such a net, it is straightforward to check that all runs of $\mathcal{M}_S$ terminate in a final state. If $\mathcal{M}_R$ has a run which is infinite (resp. deadlocks), by Lemma 4.52 the map trsf would map it into a run of $\mathcal{M}_S$ which is infinite (resp. deadlocks), which yields a contradiction.

□

Proof. (of Theorem 4.56.) Immediately follows from Lemma 4.59 above. □

Given an arbitrary net R, we of course do not know if it reduces to a normal form or not, but we are still able to use the facts above to prove that $\mathcal{M}_R$ is deadlock-free:

Theorem 4.60 (Deadlock-Freedom of the SIAM)**.** *Let R be a SMEYLL net where no ? appears in its conclusions. If a run of $\mathcal{M}_R$ terminates in a state* T*, then* T *is a final state.*

Proof. If R has no $\bot$ or no ! in its conclusions (i.e. R is simple), deadlock-freedom immediately follows from Theorem 4.59. We show that it is true also with $\bot$ or ! in the conclusions because, we can always "close" the net R into a net without creating any new deadlocks. Let $S_{A^\bot}$ be a SMEYLL proof net with a conclusion of type $A^\bot$ (and possibly other conclusions) defined as follows.

- $S_{\bot^\bot} = S_{\mathsf{1}}$ is a net consisting of a single $\mathsf{1}$ node.
- $S_{\mathsf{1}^\bot} = S_\bot$ is a net consisting of a single axiom node with conclusions of types $\mathsf{1}$ and $\bot$.
- $S_{A \parr B^\bot} = S_{A^\bot \otimes B^\bot}$ is a net obtained by connecting the conclusion of type $A^\bot$ of the net $S_{A^\bot}$ and the conclusion of type $B^\bot$ of the net $S_{B^\bot}$ by a $\otimes$ node.
- Similarly, $S_{A \otimes B^\bot} = S_{A^\bot \parr B^\bot}$ is obtained by connecting $S_{A^\bot}$ and $S_{B^\bot}$ by $\parr$ node.
- $S_{!\, A^\bot} = S_{?\, A^\bot}$ is a net obtained by connecting $?d$ node below the conclusion of type $A^\bot$ of the net $S_{?\, A^\bot}$.
- $S_{?\, A^\bot}$ is undefined.

Therefore, the conclusions of the net $S_{A^\bot}$ other than one of type $A^\bot$ are of types $X^\bot, X, 1$. Let $\overline{R}$ be the net obtained from R by connecting each conclusion of type A with the conclusion of type $A^\bot$ of the net $S_{A^\bot}$ by a cut node. Then it is straightforward to see that the machine $\mathcal{M}_R$ deadlocks if and only if the machine $\mathcal{M}_{\overline{R}}$ deadlocks. □

We stress that in the statement above there is *no assumption that the conclusions are simple formulas* (unlike in Lemma 4.59 or Theorem 4.19). The constraint that the conclusions are required not to contain the ? modality is instead an essential limit, which is intrinsic in most presentations of GoI (see e.g. [39]).

4.4 Multi $\bot$-Box

In the rest of the thesis, one main content is on how to interpret some PCF-like calculi as (appropriately extended) SMEYLL nets and multi-token machines. To do so, we have to further extend our framework to deal with two ingredients of such calculi, namely *branching* and a notion of *memory*. In this short section, we introduce a modification of $\bot$-boxes into *multi $\bot$-boxes* that allow us to interpret branching. The notion of memory is given for the deterministic, natural number case in the next section, and given as an abstract class of memories in the next chapter.

4.4.1 SMEYLL Proof Nets with Multi $\bot$-Box

Definition 4.61 (Multi $\bot$-Box)**.** A *multi $\bot$-box* is identical to $\bot$-box except that one multi $\bot$-box has *two* contents rather than one, each with the same types of conclusions as those of a $\bot$-box. It will be depicted as in Figure 4.25 hereafter.

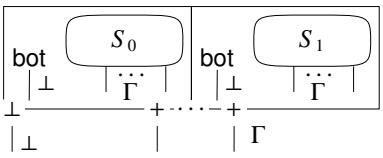


Figure 4.25: Multi $\perp$-box

Definition 4.62 (Reduction Rules for Multi $\perp$-Box)**.** The reduction rules of multi $\perp$-box is shown in Figure 4.26: if a 1 node is directly connected with the principal conclusion of the multi $\perp$-box via a cut node, it either reduces to the first content (without bot node) or reduces to the second content (without bot node).

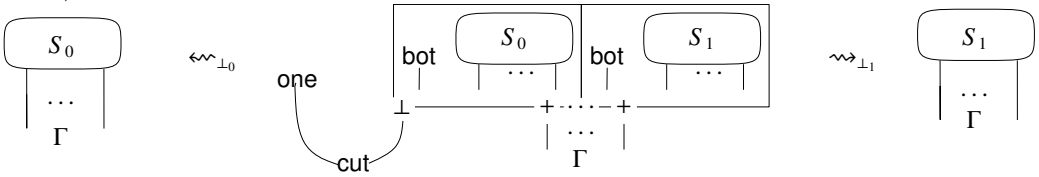


Figure 4.26: Multi $\perp$-box Reduction Rules

Note that the rules are non-deterministic: for a redex on multi $\perp$-box, either $\rightsquigarrow_{\perp_0}$ or $\rightsquigarrow_{\perp_1}$ applies. Thus as an abstract reduction system, the proof structures (or proof nets) with multi $\perp$-boxes instead of (single) $\perp$-boxes are not any more confluent; however, in the later sections the proof net system is either extended further with memories for natural numbers and become deterministic, or obtains a confluence property in a generalized sense. We do not examine its characters as a non-deterministic reduction system here, though it could be interesting.

When $\perp$-box is replaced by multi $\perp$-boxes, the definition of proof structures, of the correctness criterion, and thus of proof nets stay all the same, except we require that *both* contents of each multi $\perp$-box to be correct. On the one hand, properties of proof nets like the diamond property or confluence do not simply hold because of the reason mentioned above. On the other hand, the progress lemma and the cut elimination property do hold and the proofs of them done in Section 4.2 also apply because none of them, especially the definition of priority order, rely on the fact that a $\perp$-box contains only one content.

4.4.2 The SIAM with Multi $\perp$-Box

The transitions of the SIAM also gets non-deterministic when multi $\perp$-boxes are present, as shown in Figure 4.27. When a token arrives at the principal conclusion of a multi $\perp$-box, it enters either the first content or the second content of the box, and becomes stable. The multi-token condition is also modified. When a token $(e, \mathrm{s}, \mathrm{t})$ arrives at one of the auxiliary conclusions of a multi $\perp$-box, if $\mathrm{t} \in \mathrm{Copy}_{\mathrm{T}}(S_0)$ the token goes onto the corresponding conclusion of S_0; if $\mathrm{t} \in \mathrm{Copy}_{\mathrm{T}}(S_1)$ it goes onto S_1.
Again, the diamond property, confluence, and uniqueness of normal form do not hold, simply because the two transitions are mutually exclusive: if a token enters one content of the box it cannot move into another. This is however not problematic, since we will not use multi $\perp$-boxes alone: we will use them with a notion of *memory* and then those pleasant properties hold (as usual in section 4.5; in a generalized sense in chapter 5).

The statement of the mutual termination lemma now becomes:

Lemma 4.63 (Mutual Termination with Multi $\perp$-box)**.** *Let R be a simple SMEYLL net, possibly with multi $\perp$-boxes.*

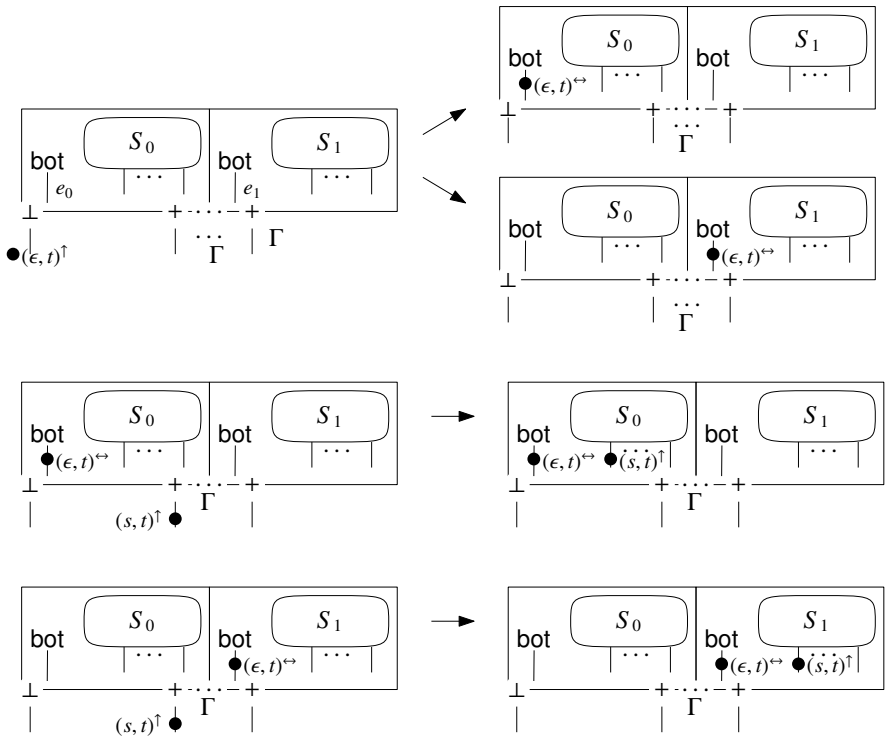


Figure 4.27: Transition on Multi ⊥-Box

1. *If* there exists *a run of $\mathcal{M}_R$ terminates, then* there exists *a sequence of reductions starting from R that terminates.*

2. *If* there exists *a sequence of reductions starting from R that terminates, then* there exists *a run of $\mathcal{M}_R$ terminating in a final state.*

The proofs stays almost the same. As a consequence, deadlock-freedom holds also with multi ⊥-box.

4.5 Interpretation of Call-by-Name and Call-by-Value PCF

In this section, we show a superiority of the SIAM compared to the usual single-token machine framework. That is, the SIAM is able to uniformly distinguish the call-by-name and call-by-value strategies of a language (here a variant of PCF) simply by the standard call-by-name and call-by-value translation into linear logic, without any additional structure or transformation such as the continuation passing style transformation [85].

The language we interpret by SMEYLL nets and SIAMs is mostly the standard PCF language. The *types* A, B and the *terms* M, N, P are defined by the following BNFs:

$$\begin{array}{rcl} \mathrm{M}, N, P & ::= & x \mid \lambda x.\mathrm{M} \mid \mathrm{MN} \mid \pi_l(\mathrm{M}) \mid \pi_r(\mathrm{M}) \\ & & \mid \langle \mathrm{M}, \mathrm{N} \rangle \mid \overline{n} \mid \mathtt{succ}(\mathrm{M}) \mid \mathtt{pred}(\mathrm{M}) \\ & & \mid \mathtt{if}\,\mathrm{P}\,\mathtt{then}\,\mathrm{M}\,\mathtt{else}\,\mathrm{N} \mid \mathtt{letrec}\, f\, x = \mathrm{M}\,\mathtt{in}\,\mathrm{N}, \\ A, B & ::= & \mathbb{N} \mid A \to B \mid A \times B. \end{array}$$

Aside from minor difference of notations, two rules are non-standard: one is the `if then else` rule in which the `then` clause and `else` are forced to be closed. This is for a technical reason regarding the translations into proof nets

$$\frac{}{\Delta, x : A \vdash x : A} \qquad \frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x.M : A \to B} \qquad \frac{\Delta \vdash M : A \to B \quad \Delta \vdash N : A}{\Delta \vdash MN : B}$$

$$\frac{\Delta \vdash M : A \times B}{\Delta \vdash \pi_l(M) : A} \qquad \frac{\Delta \vdash M : A \times B}{\Delta \vdash \pi_r(M) : B} \qquad \frac{\Delta \vdash M : A \quad \Delta \vdash N : B}{\Delta \vdash \langle M, N \rangle : A \times B}$$

$$\frac{}{\Delta \vdash \overline{n} : \mathbb{N}} \qquad \frac{\Delta \vdash M : \mathbb{N}}{\Delta \vdash \mathtt{succ}(M) : \mathbb{N}} \qquad \frac{\Delta \vdash M : \mathbb{N}}{\Delta \vdash \mathtt{pred}(M) : \mathbb{N}} \qquad \frac{\Delta \vdash P : \mathbb{N} \quad \Delta \vdash M : A \quad \Delta \vdash N : A}{\Delta \vdash \mathtt{if}\, P\, \mathtt{then}\, M\, \mathtt{else}\, N : A}$$

$$\frac{\Delta, f : A \to B, x : A \vdash M : B \quad \Delta, f : A \to B \vdash N : C}{\Delta \vdash \mathtt{letrec}\, f\, x = M\, \mathtt{in}\, N : C}$$

Table 4.1: Typing rules for PCF

that we will define later in this chapter. The expressiveness is not sacrificed by this restriction, since we can always abstract all the free variables in those terms and then provide the variables as arguments. The other non-standard one is use of the `letrec` constructor instead of the usual Y combinator. This is because we would like to consider both the call-by-name strategy and the call-by-value strategy in the same language, while the usual Y combinator works well only in the case of the call-by-name strategy: any term YM reduces to M(YM) and the argument (YM) immediately starts to diverge in the call-by-value strategy. Moreover, to explicitly force the `letrec` constructor to bind functions, it has two arguments (the function f and its argument x) instead of just one. The typing rules are shown in Table 4.1.

4.5.1 PCF Net

The notion of natural number is included in the language of PCF, while not in SMEYLL nets. Thus to interpret PCF we extend the system of SMEYLL nets with an external *memory* where natural numbers are stored, operations on memories, and decorations on nets indicating how equipped memories and operations are handled along reductions or transitions of the multi-token machine (which is also extended in the next section). From now on, we always base our discussion on the SMEYLL proof net system with $\bot$-boxes replaced by multi $\bot$-boxes.

Definition 4.64 (Natural Number Memory)**.** Let $\mathcal{I}$ be a countably infinite set and $\mathcal{L} = \{\mathtt{max}, \mathtt{succ}, \mathtt{pred}\}$ equipped with a function arity$\colon \mathcal{L} \to \mathbb{N}$ mapping arity($\mathtt{max}$) = 2 and arity($\mathtt{succ}$) = arity($\mathtt{pred}$) = 1. A *natural number memory* m is a function $\mathrm{m} \colon \mathcal{I} \to \mathbb{N}$ together with the following two maps[4] (where Mem denotes the set of memories):

- test$\colon \mathcal{I} \times \mathrm{Mem} \to \mathbb{B} \times \mathrm{Mem}$, defined as $\mathrm{test}(i, \mathrm{m}) = (\mathtt{true}, \mathrm{m})$ if $\mathrm{m}(i) = 0$ and as $\mathrm{test}(i, \mathrm{m}) = (\mathtt{false}, \mathrm{m})$ if $\mathrm{m}(i) \neq 0$. [5]
- update$\colon \mathcal{I}^* \times \mathcal{L} \times \mathrm{Mem} \rightharpoonup \mathrm{Mem}$, where $\mathrm{update}((i, j), \mathtt{max}, \mathrm{m})$ is defined to be $\mathrm{m}[i \mapsto \max(\mathrm{m}(i), \mathrm{m}(j)), j \mapsto \max(\mathrm{m}(i), \mathrm{m}(j))]$, $\mathrm{update}(i, \mathtt{succ}, \mathrm{m})$ is defined to be $\mathrm{m}[i \mapsto (\mathrm{m}(i) + 1)]$, $\mathrm{update}(i, \mathtt{pred}, \mathrm{m})$ is defined to be $\mathrm{m}[i \mapsto (\max(\mathrm{m}(i) - 1), 0)]$, and otherwise undefined.

[4] In [18], we also included a map init. It is no more than a design choice and does not make any difference.

[5] The codomain includes Mem for a generalization that will be done later in the thesis. One may ignore it in this chapter.

We also need some auxiliary sets of nodes below to define the translation.

Definition 4.65 (SyncNode and SurfOne)**.** Given a SMEYLL proof net R, The set $\mathrm{SyncNode}_R$ consists of all the sync nodes in R (including those inside a box). The set $\mathrm{SurfOne}_R$ consists of all the 1 nodes at surface of R (i.e. excluding those inside a box).

Definition 4.66 (Input)**.** Given a SMEYLL net R, the set Input_R is defined to be the union of the set of all conclusions of 1 nodes at surface of R and the set of all occurrence of $\bot$ in the conclusions of R.

Definition 4.67 (Decorated Proof Net)**.** A *decorated SMEYLL proof net* is a pair (R, op_R) where R is a SMEYLL proof net and $\mathrm{op}_R \colon \mathrm{SyncNode}_R \to \mathcal{L}$ is a function satisfying that $\mathrm{arity}(s)$ is equal to the number of 1's occurring in the conclusions of s for each sync node s.

Definition 4.68 (PCF Net)**.** A *PCF net* is a triple $\mathbf{R} = ((R, \mathrm{op}_R), \mathrm{ind}_R, \mathrm{m}_R)$ where (R, op_R) is a decorated SMEYLL proof net, $\mathrm{ind}_R \colon \mathrm{SurfOne}_R \rightharpoonup \mathcal{I}$ is a partial injective function, and $\mathrm{m}_R \in \mathrm{Mem}$.

Notation 4.69 (Pictorial Representation of PCF Net)**.** A PCF net $\mathbf{R} = ((R, \mathrm{op}_R), \mathrm{ind}_R, \mathrm{m}_R)$ will be hereafter represented as shown in Figure **??**. A sync node a with $\mathrm{op}_R(a) = l$ is depicted as a sync node with the name l next to the node; a 1 node a' with $\mathrm{ind}_R(a') = i$ is depicted as a 1 node connected to the index i by a dotted line; and the memory m_R is shown in a rectangle put next to the SMEYLL net.

Definition 4.70 (Reduction Rules of PCF Nets)**.** Most of the reduction rules of PCF nets are the "same" as those of SMEYLL nets: given a rule of SMEYLL nets in Figure 4.28, it is extended to a reduction rule of PCF nets that defines a reduction $((R, \mathrm{op}), \mathrm{ind}, \mathrm{m}) \rightsquigarrow ((S, \mathrm{op}), \mathrm{ind}, \mathrm{m}$ in PCF nets, where $R \rightsquigarrow S$ in SMEYLL nets by that rule.

Rules in Figure 4.29 are specific to PCF nets. The rule $\mathrm{link}(i)$ reduces $((R, \mathrm{op}_R), \mathrm{ind}_R, \mathrm{m}_R)$ to $((R, \mathrm{op}_R), \mathrm{ind}_R \cup \{a \mapsto i\}, \mathrm{m}_R)$ where a is the 1 node in the redex and i is a fresh index. The rule update reduces $((R, \mathrm{op}_R), \mathrm{ind}_R, \mathrm{m}_R)$ to $((S, \mathrm{op}_R), \mathrm{ind}_R, \mathrm{update}(i_1, \ldots, i_k, l, \mathrm{m}_R))$ where S is the net obtained by the rule and $i_1, \ldots, i_k$ are the indexes as shown in the figure. The rule $\mathrm{test}(i)$ reduces the net, either $((R, \mathrm{op}_R), \mathrm{ind}_R, \mathrm{m}_R)$ to $((S_0, \mathrm{op}_{S_0}), \mathrm{ind}_R \setminus \{a \mapsto i\}, \mathrm{m}_R)$ or to $((S_1, \mathrm{op}_{S_1}), \mathrm{ind}_R \setminus \{a \mapsto i\}, \mathrm{m}_R)$ depending on the value $\mathrm{test}(i, \mathrm{m}_R)$, where S_i is the net obtained by replacing the redex by S_i in the figure and op_{S_i} is obtained by restricting op_R to op_{S_i} accordingly.

Definition 4.71 (Convergence to n)**.** Let $\mathbf{R}$ be a PCF net with the only conclusion 1. We write $\mathbf{R} \Downarrow n$ if there exists a normal PCF net $\mathbf{S} = ((S, \mathrm{op}_S), \mathrm{ind}_S, \mathrm{m}_S)$ whose underlying SMEYLL proof net consists of a single 1 node a, the PCF net $\mathbf{S}$ satisfies $\mathbf{R} \rightsquigarrow^* \mathbf{S} \not\rightsquigarrow$, and $\mathrm{m}_S(\mathrm{ind}_S(a)) = n$.

4.5.2 PCF Synchronous Interaction Abstract Machine

To interpret PCF nets, the SIAM is also extended with the notion of natural number memory, in a way analogous to what is done for SMEYLL proof nets: each state is now equipped with a memory, and several transition rules now modifies the memory attached to states. The resulting multi-token machine will be called *PCF Synchronous Interaction Abstract Machine (PSIAM)*.

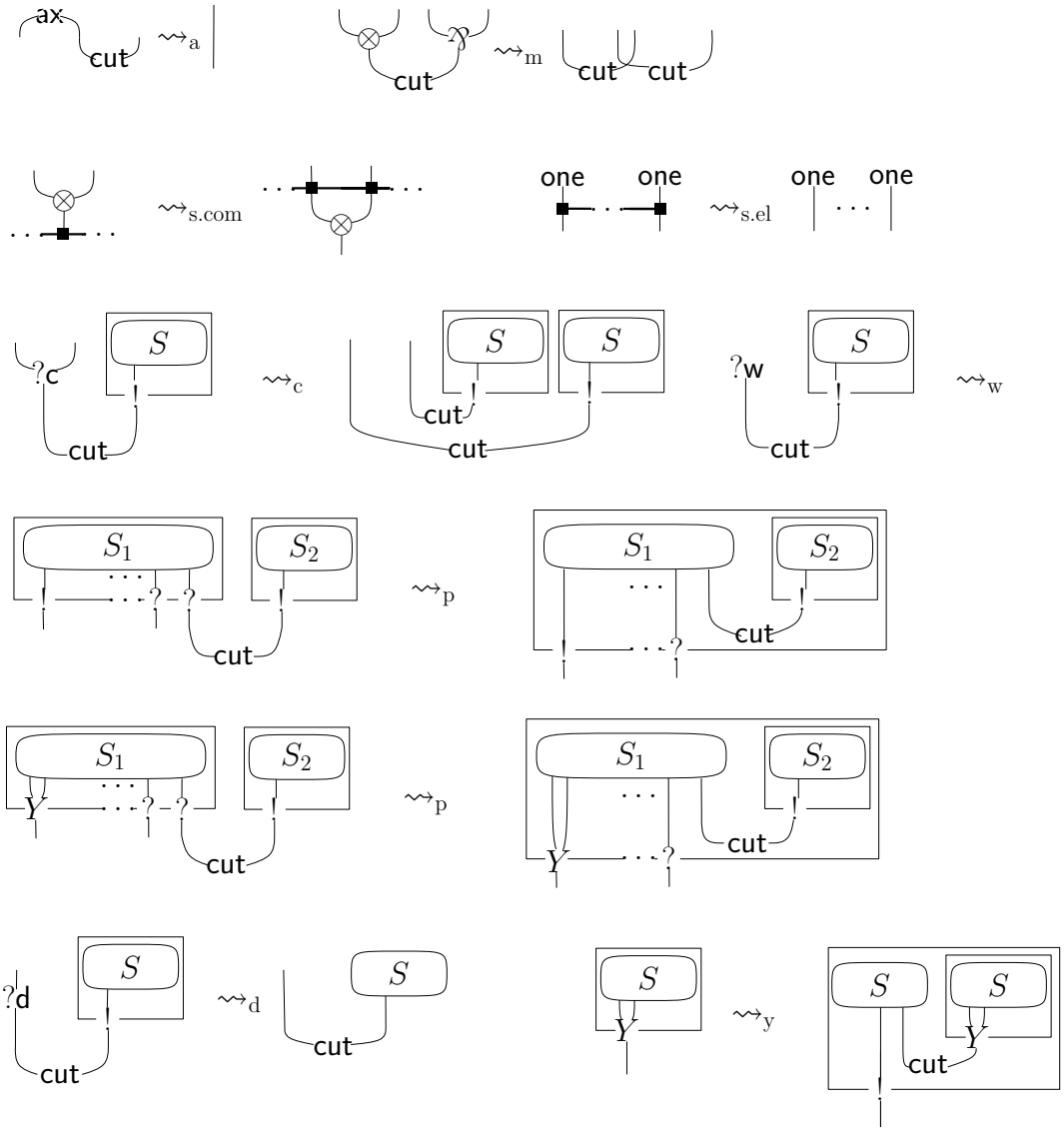
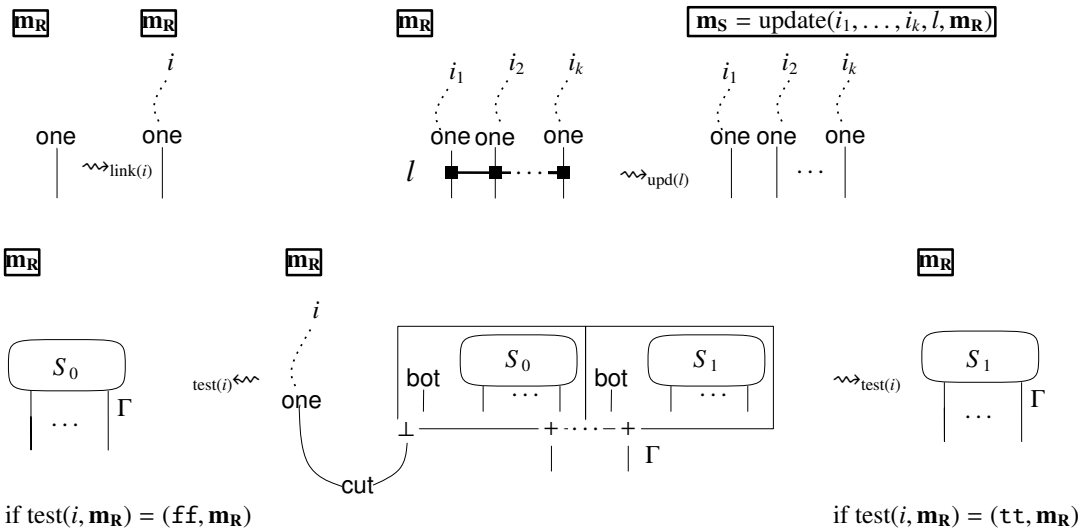


Figure 4.28: Reduction Rules of PCF Nets



if test($i, \mathbf{m_R}$) = (ff, $\mathbf{m_R}$) if test($i, \mathbf{m_R}$) = (tt, $\mathbf{m_R}$)

Figure 4.29: Additional Reduction Rules of PCF Nets

Definition 4.72 (PSIAM State)**.** Let $\mathbf{R} = ((R, \mathrm{op}_R), \mathrm{ind}_R, \mathrm{m}_R)$ be a PCF net. A *PSIAM state* over $\mathbf{R}$ is a triple $\mathrm{T} = (\mathrm{T}, \mathrm{ind}_\mathrm{T}, \mathrm{m}_\mathrm{T})$ where T is a state of the SIAM $\mathcal{M}_R$ (of the SMEYLL net R), ind_T is a partial function $\mathrm{ind}_\mathrm{T} \colon \mathrm{Start}_R \rightharpoonup \mathcal{I}$ that is however total on Init_R, and m_R is a memory in Mem, satisfying that if $\mathrm{orig}(\mathbf{p}) = \mathbf{q}$ for a position $\mathbf{p} \in \mathrm{T}$ immediately below a 1 node a, then $\mathrm{ind}_\mathrm{T}(\mathbf{q})$ is defined. The set of PSIAM states over a PCF net $\mathbf{R}$ is denoted by $\mathcal{S}_\mathbf{R}$. The initial state $\mathbf{I}_\mathbf{R}$ is give by $(\mathbf{I}_R, \mathrm{ind}_{\mathbf{I}_\mathbf{R}}, \mathrm{m}_R)$ satisfying $\mathrm{ind}_{\mathbf{I}_\mathbf{R}}(\mathbf{p}) = \mathrm{ind}_\mathbf{R}(a)$ where $\mathbf{p}$ is the position immediately below the 1 node $a \in \mathrm{SurfOne}_R$ and undefined on any other position.

Definition 4.73 (Transition Rules of the PSIAM)**.** Given a PCF net $\mathbf{R}$, the transition rules of the PSIAM is defined to be a relation $\to \subseteq \mathcal{S}_\mathbf{R} \times \mathcal{S}_\mathbf{R}$. Most of the transition rules are essentially the same as those of the SIAM: $(\mathrm{T}, \mathrm{ind}, \mathrm{m}) \to (\mathrm{U}, \mathrm{ind}, \mathrm{m})$ holds if $\mathrm{T} \to \mathrm{U}$ as an SIAM transition, except the following two cases.

- A transition $(\mathrm{T}, \mathrm{m}_0) \to (\mathrm{U}, \mathrm{m}_1)$ crossing a sync node a now modifies the memory m_0 to m_1, where $\mathrm{m}_1 = \mathrm{update}(i_1, \ldots, i_k, \ell, \mathrm{m}_0)$ for $\ell = \mathrm{op}_\mathbf{R}(a)$ and $\mathrm{k} = \mathrm{arity}(\ell)$.

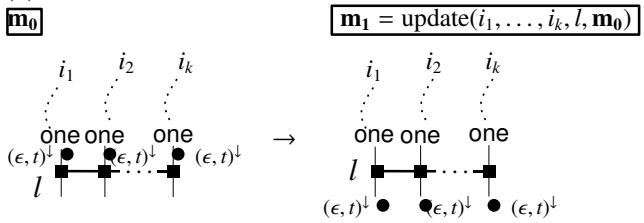


Figure 4.30: PSIAM Transition on sync Node

- Let b be a multi $\perp$-box as in Fig. 4.31, where S_0 and S_1 are the two contents of the box, and the edges e_0, e_1 are as indicated in the figure. When a token is in position $\mathbf{p} = (e, \epsilon, \mathrm{t})$ on the principal conclusion of the box, it moves to $(e_0, \epsilon, \mathrm{t})$ if $\mathrm{test}(\mathrm{orig}(\mathbf{p}), \mathrm{m})$ returns the boolean `ff` (arrow (i) in Fig. 4.31) and it moves to $(e_1, \epsilon, \mathrm{t})$ if $\mathrm{test}(\mathrm{orig}(\mathbf{p}), \mathrm{m})$ returns `tt` (arrow (ii) in Fig. 4.31).

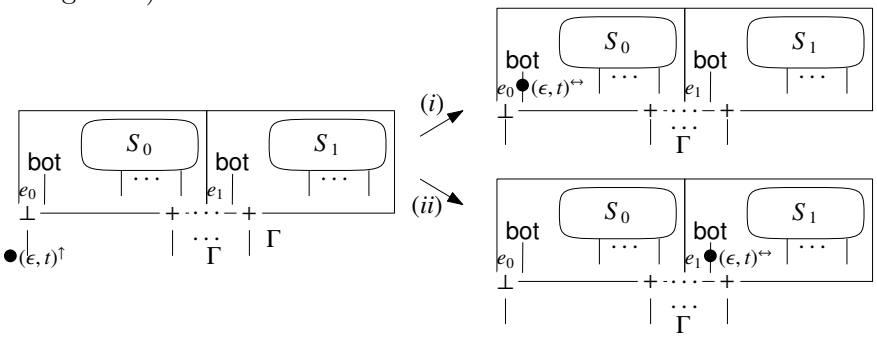


Figure 4.31: PSIAM Transition on Multi $\perp$-Box

Definition 4.74 (PCF Synchronous Interaction Abstract Machine)**.** Given a PCF net $\mathbf{R}$, the *PCF Synchronous Interaction Abstract Machine (PSIAM)* $\mathcal{M}_\mathbf{R}$ is the abstract rewriting system $(\mathcal{S}_\mathbf{R}, \to)$.

Definition 4.75. Similarly to PCF nets, we write $\mathcal{M}_\mathbf{R} \Downarrow n$ if $\mathcal{M}_\mathbf{R}$ terminates in its unique final state with the only content of the memory in the final state is n.

4.5.3 Adequacy

The definitions of PCF nets and the PSIAM lead to an adequacy result similar to the one for SMEYLL nets and the SIAM. It is even finer in the sense that it

does not only state on mere convergence of the two systems, but also takes into account the contents of the memories attached to nets and states.

Theorem 4.76 (Adequacy between PCF Nets and PSIAM)**.** *Let* $\mathbf{R}$ *be a PCF net with the only conclusion of type* 1*. Then* $\mathbf{R} \Downarrow n$ *if and only if* $\mathcal{M}_{\mathbf{R}} \Downarrow n$*.*

The proof technique to show Theorem 4.76 is also an adaptation of the one for the proof of adequacy between SMEYLL nets and the SIAM. First, the correspondence of positions by the transformation map remains the same except the case of $\perp$ reduction as shown in Figure 4.32, where $\mathbf{R} \rightsquigarrow \mathbf{R}_0$ if $\mathrm{test}(i) = 0$ on the index i of the 1 node in the redex, and $\mathbf{R} \rightsquigarrow \mathbf{R}_1$ if $\mathrm{test}(i) \neq 0$.

Then, in order to take care of memories cleanly, we restrict the domain of the transformation map trsf to a set [trsf] of PSIAM states, given by the following definition.

Definition 4.77. Let $\mathbf{R} \rightsquigarrow \mathbf{S}$ be a PCF net reduction. The set $[\mathrm{trsf}] \subseteq \mathcal{S}_{\mathbf{R}}$ is defined as follows, depending on the reduction rule.

- If the reduction is by $\rightsquigarrow_{\mathrm{link}(j)}$ rule, we define [trsf] as the set of states in which $\mathrm{ind}(\mathbf{p}) = j$, where $\mathbf{p} \in \mathrm{SurfOne}_R$ is the position associated to the 1 node a.
- If the reduction is by $\rightsquigarrow_{\mathrm{update}(s)}$ rule, assume $\mathbf{p}_1, \ldots, \mathbf{p}_n$ are the positions associated to the premises of the sync node s in the redex. (observe that each $\mathbf{p}_i$ belongs to Ones_R). [trsf] is defined to be the set of states $(\mathrm{T}, \mathrm{ind}_{\mathrm{T}}, \mathrm{m}_{\mathrm{T}})$ satisfying $\{\mathbf{p}_1, \ldots, \mathbf{p}_n\} \subseteq \mathrm{orig}(T)$ and $\{\mathbf{p}_1, \ldots, \mathbf{p}_n\} \not\subseteq \mathrm{T}$.
- If the reduction is by $\rightsquigarrow_{\mathrm{test}(j)}$ rule, we define [trsf] as the set of states that contain a token on the left $\perp$ node of the $\perp$-box in the redex (the edge e_0 in Figure 4.32) if $\mathrm{test}(j) = (\mathtt{true}, \mathrm{m})$; if $\mathrm{test}(j) = (\mathtt{false}, \mathrm{m})$, then [trsf] is defined to be the set of states containing a token on the right $\perp$ node.
- Otherwise we define $[\mathrm{trsf}] = \mathcal{S}_{\mathbf{R}}$.

Definition 4.78 (Transformation Map)**.** Let $\mathbf{R} \rightsquigarrow \mathbf{S}$ be a reduction on a redex r in the PCF net $\mathbf{R}$. The *transformation map* $\mathrm{trsf}_{\mathbf{R},r,\mathbf{S}} \colon [\mathrm{trsf}] \to \mathcal{S}_{\mathbf{S}}$ is defined by

$$\mathrm{trsf}(\mathrm{T}, \mathrm{ind}_{\mathrm{T}}, \mathrm{m}_{\mathrm{T}}) = (\mathrm{trsf}(\mathrm{T}), \mathrm{ind}_{\mathrm{T}}, \mathrm{m}_{\mathrm{T}})$$

where trsf on the right-hand side of equation is the transformation map for the SIAM states.

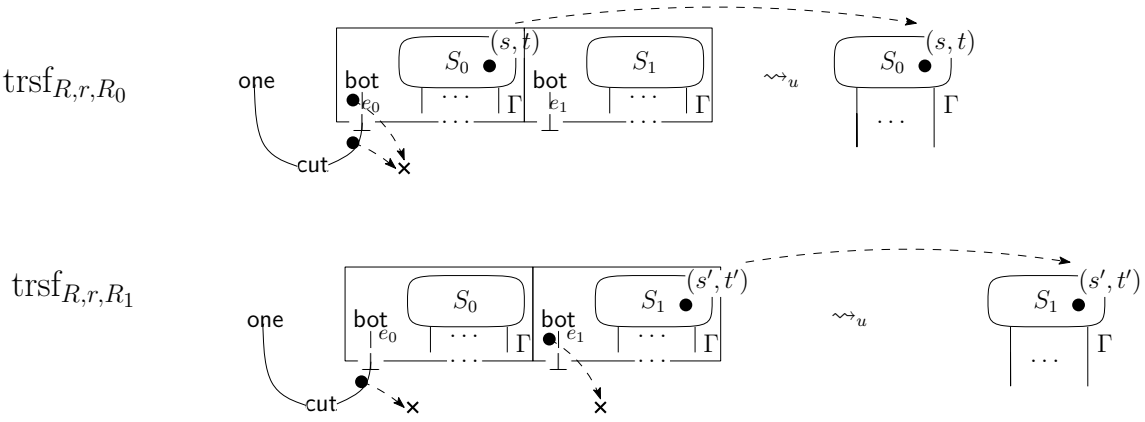


Figure 4.32: Transformation Map on a Multi $\perp$-box

Remark 4.79. Consider the following sequence of transitions from the initial state $\mathbf{I}$, and let $\mathbf{S}$ be the resulting state. Observe that $\mathbf{S} \in [\mathrm{trsf}]$.

- If the reduction is s.el, first create one token per each 1 node in the redex, then apply the transition rule to cross the sync node in the redex.
- If the reduction is $\bot$, first create one token on the 1 node in the redex, make the token created cross the cut node in the redex, then apply the transition rule to enter the $\bot$-box in the redex.

Even if $\mathbf{I} \to^* \mathbf{T} \notin [\mathrm{trsf}]$, by confluence of the PSIAM there exists $\mathbf{T}'$ satisfying $\mathbf{T} \to^* \mathbf{T}'$ and $\mathbf{S} \to^* \mathbf{T}'$. Observe that if a state is in $[\mathrm{trsf}]$, then any state reachable from that state is also in $[\mathrm{trsf}]$. Thus $\mathbf{T} \to^* \mathbf{T}' \in [\mathrm{trsf}]$. Therefore we can always assume that a state in a run is in $[\mathrm{trsf}]$ when we analyze convergence, divergence, or deadlock of the machine.

Then, properties analogous to Lemma 4.52 also hold:

Lemma 4.80. *Let* $\mathbf{R}$ *be a PCF net. For any reduction* $\mathbf{R} \rightsquigarrow \mathbf{R}'$*,*

1. *If* $\mathrm{T} \to \mathrm{U}$ *in* $\mathcal{M}_{\mathbf{R}}$ *then* $\mathrm{trsf}(\mathrm{T}) \to^* \mathrm{trsf}(\mathrm{U})$ *in* $\mathcal{M}_{\mathbf{R}'}$*.*
2. *If* T *is an initial state in* $\mathcal{M}_{\mathbf{R}}$*, then so is* $\mathrm{trsf}(\mathrm{T})$ *in* $\mathcal{M}_{\mathbf{R}'}$*.*
3. *If* T *is a final state in* $\mathcal{M}_{\mathbf{R}}$*, then so is* $\mathrm{trsf}(\mathrm{T})$ *in* $\mathcal{M}_{\mathbf{R}'}$*.*
4. *If* T *is a deadlock state in* $\mathcal{M}_R$*, then so is* $\mathrm{trsf}(\mathrm{T})$ *in* $\mathcal{M}_S$*.*

Proof. Each statement can be proved by case analysis.

1. If the transition $\mathrm{T} \to \mathrm{U}$ is not on the redex of $R \rightsquigarrow S$ then the claim holds, because the positions of tokens and the structure are the same except around the redex. For the reductions except $\rightsquigarrow_{s.el}$, $\rightsquigarrow_{bot.el}$ and $\rightsquigarrow_{decor}$ the proof is the same as that in the SIAM case since memories are not modified by the reduction and trsf, and mapping of positions is the same as the SIAM case. For those three rules of reduction relevant to memories:
   - $\underline{\rightsquigarrow_{\mathrm{s.el}}}$ We only have to consider the transition $\mathrm{T} \to \mathrm{U}$ that crosses the sync node reduced; by definition of the transformation $\mathrm{trsf}(\mathrm{T}) = \mathrm{trsf}(\mathrm{U})$ holds.
   - $\underline{\rightsquigarrow_{\bot}}$ If the transition crosses the cut node or enters the box then $\mathrm{trsf}(\mathrm{T}) = \mathrm{trsf}(\mathrm{U})$ by definition. Since no transition can modify the value of the memory pointed by the 1 node, the contents chosen by the reduction and by the transitions coincide. Thus for all transitions $\mathrm{T} \to \mathrm{U}$ in the box $\mathrm{trsf}(\mathrm{T}) \to \mathrm{trsf}(\mathrm{U})$ holds.
   - $\underline{\rightsquigarrow_{\mathrm{decor}}}$ Since the machines $\mathcal{M}_{\mathbf{R}}$ and $\mathcal{M}_{\mathbf{R}'}$ are the same by definition,for every transition $\mathrm{T} \to \mathrm{U}$ we have $\mathrm{trsf}(\mathrm{T}) \to \mathrm{trsf}(\mathrm{U})$.
2. Almost the same as the SIAM: in the case of $\rightsquigarrow_{\mathrm{s.el}}$, the memory in the initial state in $\mathcal{M}_{\mathbf{R}}$ is mapped to that in $\mathcal{M}_{\mathbf{R}'}$ by definition of trsf.

3.–5. Same as the SIAM, since the definition of terminal/final/deadlock state does not rely on memories.

□

Lemma 4.81. *Let* $\mathbf{R} \rightsquigarrow \mathbf{S}$ *be a PCF net reduction. The function* trsf *maps each run from a state* $\mathbf{T} \in [\mathrm{trsf}]$ *of* $\mathcal{M}_{\mathbf{R}}$ *into a run of* $\mathcal{M}_{\mathbf{S}}$ *which converges, diverges, or deadlocks iff the run of* $\mathcal{M}_{\mathbf{R}}$ *does.*

Proof. The proof proceeds in exactly the same way as that of the SIAM thanks to Lemma 4.80. □

Fact 4.82. Let $\mathbf{R}$ be a PCF net. The following holds by definition of trsf:

- Let $\mathbf{R} \rightsquigarrow \mathbf{R}'$ be not a update reduction and (T, m) be a state of $\mathcal{M}_{\mathbf{R}}$. Then $\mathrm{m}(\mathrm{ind}_{\mathbf{R}'}(\mathrm{orig}_S(\mathrm{trsf}(\mathbf{p})))) = \mathrm{m}(\mathrm{ind}_{\mathbf{R}}(\mathrm{orig}_R(\mathbf{p})))$.
- Let $\mathbf{R} \rightsquigarrow \mathbf{R}'$ be a update reduction. Then
  $\mathrm{m}_i(\mathrm{ind}_{\mathbf{R}'}(\mathrm{orig}_S(\mathrm{trsf}(\mathbf{p})))) = \mathrm{m}_i(\mathrm{ind}_{\mathbf{R}}(\mathrm{orig}_R(\mathbf{p})))$ if $\mathbf{T} \in [\mathrm{trsf}]$.

Intuitively speaking, these facts mean that the content of the memory associated to a token is preserved by trsf map.

Moreover, similarly to what we have seen in Section 4.3.4,

Lemma 4.83. *Let* $\mathbf{R}$ *be a PCF net where all conclusions have type* 1*. The machine* $\mathcal{M}_{\mathbf{R}}$ *terminates in a final state* $(\mathrm{T}, \mathrm{ind}_{\mathrm{T}}\mathrm{m}_{\mathrm{T}})$ *if and only if* $\mathbf{R}$ *reduces to a cut and sync free net (say* $\mathbf{S} = ((S, \mathrm{op}_S), \mathrm{ind}_S, \mathrm{m}_S)$*). Moreover,*

$$\mathrm{m}_S = \mathrm{m}'_{\mathrm{T}}$$

where m'_{T} *is the restriction of* m *to the elements pointed to by original positions of final positions in* T*.*

Proof. if. Every run in the cut and sync free net converges because such a net with all conclusions typed by 1 is nothing but an MLL proof net. Hence by Lem. 4.81 any run in $\mathcal{M}_{\mathbf{R}}$ also converges.
only if. A similar argument as in the case of the SIAM applies: if $\mathcal{M}_{\mathbf{R}}$ terminates then the reduction of $\mathbf{R}$ terminates, and the normal form is cut and sync free. We observe that the counterpart of Theorem 4.19 holds for PCF nets, because the number of contents of a $\perp$-box is irrelevant to the proof of the theorem, and memories do not affect the possibility of a reduction.
$\mathrm{m}_S = \mathrm{m}'_{\mathrm{T}}$. Follows from Fact 4.82. □

Theorem 4.76 follows from the lemma above.

Proof. (of Theorem 4.76) Suppose $\mathbf{R} \Downarrow n$. By definition, there exists a cut and sync free PCF net $\mathbf{S}$ (of which underlying SMEYLL net consists of a single 1 node). Thus by Lemma 4.83, $\mathcal{M}_{\mathbf{R}}$ terminates in a final state $(\mathrm{T}, \mathrm{m}_{\mathrm{T}})$ with
□

4.5.4 Call-by-Name Translation

The definitions and properties of PCF nets and the PSIAM being stated, we first look at the call-by-name case that is the standard one in the context of GoI. To do so, we recall the call-by-name PCF. A *call-by-name reduction context* $C[-]$ is defined by the following BNF:

$$\begin{array}{rcl} C[-] & ::= & [-] \mid C[-]N \mid \pi_l C[-] \mid \pi_r C[-] \mid \\ & & \mathtt{succ}(C[-]) \mid \mathtt{pred}(C[-]) \mid \mathtt{if}\, C[-]\, \mathtt{then}\, M\, \mathtt{else}\, N. \end{array}$$

The set of evaluation rules is the usual one shown in Table 4.2.

(1) *Axiom rules.*

$$\frac{}{(\lambda x.M)N \to_{\mathrm{cbn}} M\{x := N\}} \qquad \frac{}{\pi_l\langle M, N\rangle \to_{\mathrm{cbn}} M} \qquad \frac{}{\pi_r\langle M, N\rangle \to_{\mathrm{cbn}} N}$$

$$\frac{}{\mathtt{succ}(\overline{n}) \to_{\mathrm{cbn}} \overline{n+1}} \qquad \frac{}{\mathtt{pred}(\overline{n+1}) \to_{\mathrm{cbn}} \overline{n}} \qquad \frac{}{\mathtt{pred}(\overline{0}) \to_{\mathrm{cbn}} \overline{0}}$$

$$\frac{}{\mathtt{if}\,\overline{0}\,\mathtt{then}\,M\,\mathtt{else}\,N \to_{\mathrm{cbn}} M} \qquad \frac{}{\mathtt{if}\,\overline{n+1}\,\mathtt{then}\,M\,\mathtt{else}\,N \to_{\mathrm{cbn}} N}$$

$$\frac{}{\mathtt{letrec}\,f\,x = M\,\mathtt{in}\,N \to_{\mathrm{cbn}} N\{f := \lambda x.\mathtt{letrec}\,f\,x = M\,\mathtt{in}\,f\,x\}}$$

(2) *Congruence rules.* Provided that $C[-]$ is a call-by-name context:

$$\frac{M \to_{\mathrm{cbn}} N}{C[M] \to_{\mathrm{cbn}} C[N]}$$

Table 4.2: Call-by-name evaluation strategy for PCF

Then we give the *call-by-name translation* from the types and the language of PCF into SMEYLL types and PCF nets, which is also based on the standard one (e.g. by Girard [38]).

Definition 4.84. The *call-by-name translation* from a PCF type A to a SMEYLL formula A^* is given by:

- $\mathbb{N}^* := \mathsf{1}$
- $(A \to B)^* := ?\,(A^*)^\perp \parr B^*$
- $(A \times B)^* := !(A^*) \otimes !(B^*)$

Abusing notation, the *call-by-name translation* $(-)^*$ from PCF derivation trees to PCF nets is given in Figure 4.33. The translation of a derivation tree of the judgment $\Delta \vdash \mathrm{M} : A$ is (again abusing notation) denoted by M^*.

In general, the call-by-name translation of a derivation of the judgment $x_1 \colon A_1, \ldots, x_1 \colon A_1 \vdash \mathrm{M} : B$ is in the form shown in Figure 4.34.The following is immediate by definition of the translation:

Fact 4.85. Let M be a closed term of type $\mathbb{N}$. Then its translation M^* is a PCF net with conclusion 1.

The adequacy result—which is one of the main results of the chapter—states that the behavior of a PCF term is precisely captured by the PCF net translated from the term. Since we already obtained adequacy result for PCF nets and the PSIAM (Theorem 4.76), the PSIAM also exhibits the same behavior as the term. Hence we obtain a multi-token machine that executes the computation of PCF. Note also that the statement is bidirectional (if and only if): by contraposition, divergence is also captured by PCF nets and the PSIAM.

Theorem 4.86 (Adequacy between PCF Term and PCF Net)**.** *Let* M *be a closed term of type* $\mathbb{N}$*. Then* $\mathrm{M} \Downarrow \overline{n}$ *if and only if* $\mathrm{M}^* \Downarrow n$*.*

The proof is deferred to Section 4.5.6.

Corollary 4.87 (Adequacy between PCF Term and PSIAM)**.** *Let* M *be a closed term of type* $\mathbb{N}$*. Then* $\mathrm{M} \Downarrow \overline{n}$ *if and only if* $\mathcal{M}_{\mathrm{M}^*} \Downarrow n$*.*

Proof. Follows from Theorem 4.76 and Theorem 4.86. □

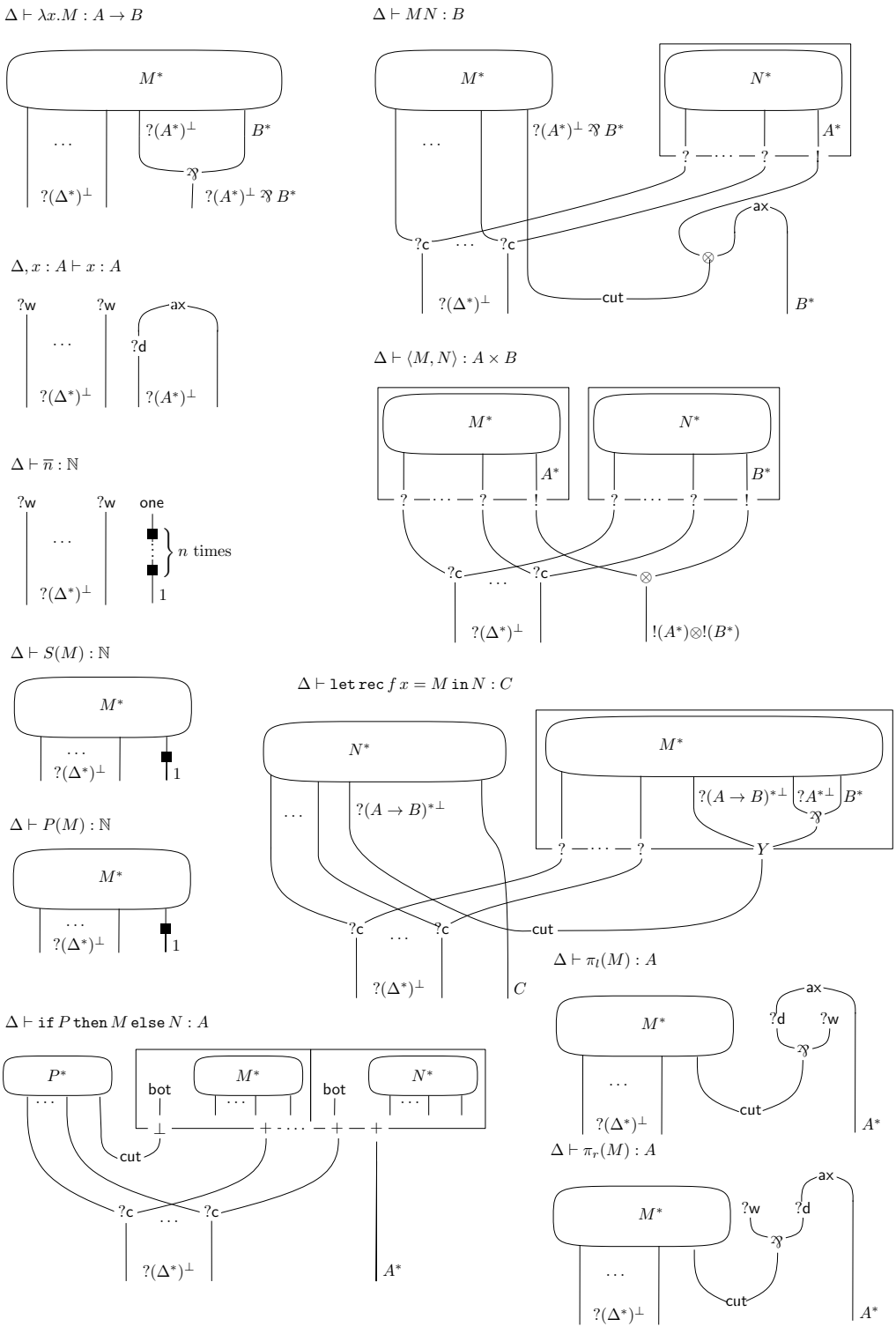


Figure 4.33: Call-by-Name Translation of PCF

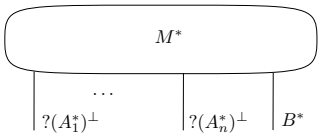


Figure 4.34: General Form of Call-by-Name Translation

(1) *Axiom rules.*

$$\frac{}{(\lambda x.M)U \to_{\mathrm{cbv}} M\{x := U\}} \qquad \frac{}{\pi_l\langle U, V\rangle \to_{\mathrm{cbv}} U} \qquad \frac{}{\pi_r\langle U, V\rangle \to_{\mathrm{cbv}} V}$$

$$\frac{}{\mathtt{succ}(\overline{n}) \to_{\mathrm{cbv}} \overline{n+1}} \qquad \frac{}{\mathtt{pred}(\overline{n+1}) \to_{\mathrm{cbv}} \overline{n}} \qquad \frac{}{\mathtt{pred}(\overline{0}) \to_{\mathrm{cbv}} \overline{0}}$$

$$\frac{}{\mathtt{if}\,\overline{0}\,\mathtt{then}\,M\,\mathtt{else}\,N \to_{\mathrm{cbv}} M} \qquad \frac{}{\mathtt{if}\,\overline{n+1}\,\mathtt{then}\,M\,\mathtt{else}\,N \to_{\mathrm{cbv}} N}$$

$$\frac{}{\mathtt{letrec}\,f\,x = M\,\mathtt{in}\,N \to_{\mathrm{cbv}} N\{f := \lambda x.\mathtt{letrec}\,f\,x = M\,\mathtt{in}\,f\,x\}}$$

(2) *Congruence rules.* Provided that $C[-]$ is a call-by-value context:

$$\frac{M \to_{\mathrm{cbn}} N}{C[M] \to_{\mathrm{cbn}} C[N]}$$

Table 4.3: Call-by-value evaluation strategy for PCF.

4.5.5 Call-by-Value Translation

What we showed in the previous section is however not very surprising, although such work with a multi-token machine is novel. The call-by-name strategy is known to be natural for GoI, and after all Mackie's GoI machine [69] does essentially the same thing. What is truly novel here appears in this section: we obtain analogous adequacy results also for the *call-by-value* PCF, solely by changing the translation into PCF nets, with exactly the same multi-token machine. Moreover the translation is (except some encodings to handle natural numbers) the so-called "efficient encoding" by Girard [38], without any extra construction such as continuation passing style transformation. Therefore we claim that our multi-token machine *uniformly* serves as an abstract machine both for the call-by-name and the call-by-value strategy with standard encodings, which is the first one as far as the author knows.

As in the previous section we first describe the call-by-value strategy of PCF. A *value* V is defined by the following BNF:

$$V \quad ::= \quad x \;\mid\; \lambda x.M \;\mid\; \langle U, U\rangle \;\mid\; \overline{n}.$$

A *call-by-value evaluation context* $C[-]$ is defined by:

$$\begin{aligned} C[-] \quad ::= \quad & [-] \mid C[-]N \mid VC[-] \mid \langle C[-], N\rangle \mid \langle V, C[-]\rangle \mid \\ & \pi_l C[-] \mid \pi_r C[-] \mid \mathtt{succ}(C[-]) \mid \mathtt{pred}(C[-]) \mid \\ & \mathtt{if}\,C[-]\,\mathtt{then}\,M\,\mathtt{else}\,N, \end{aligned}$$

and the call-by-value evaluation rule is shown in 4.3.

Similarly to the call-by-name case, we translate PCF type derivation into PCF nets. However, the construction gets more complicated due to the encoding that requires us to copy the content of a memory linked to a 1 node *at surface*. In order to deal with the problem, we need several auxiliary constructions in Figure 4.35, Figure 4.36, and Figure 4.37.

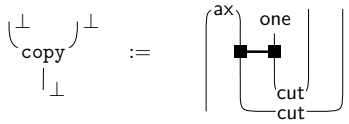


Figure 4.35: Copying Node

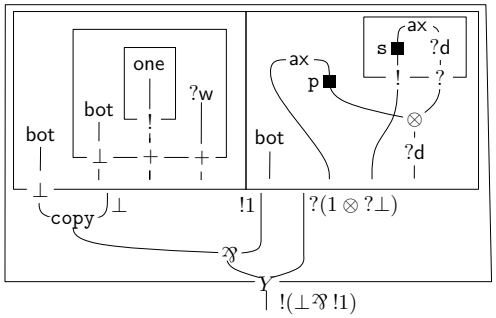


Figure 4.36: PCF Net Computing $\bot \parr !1$

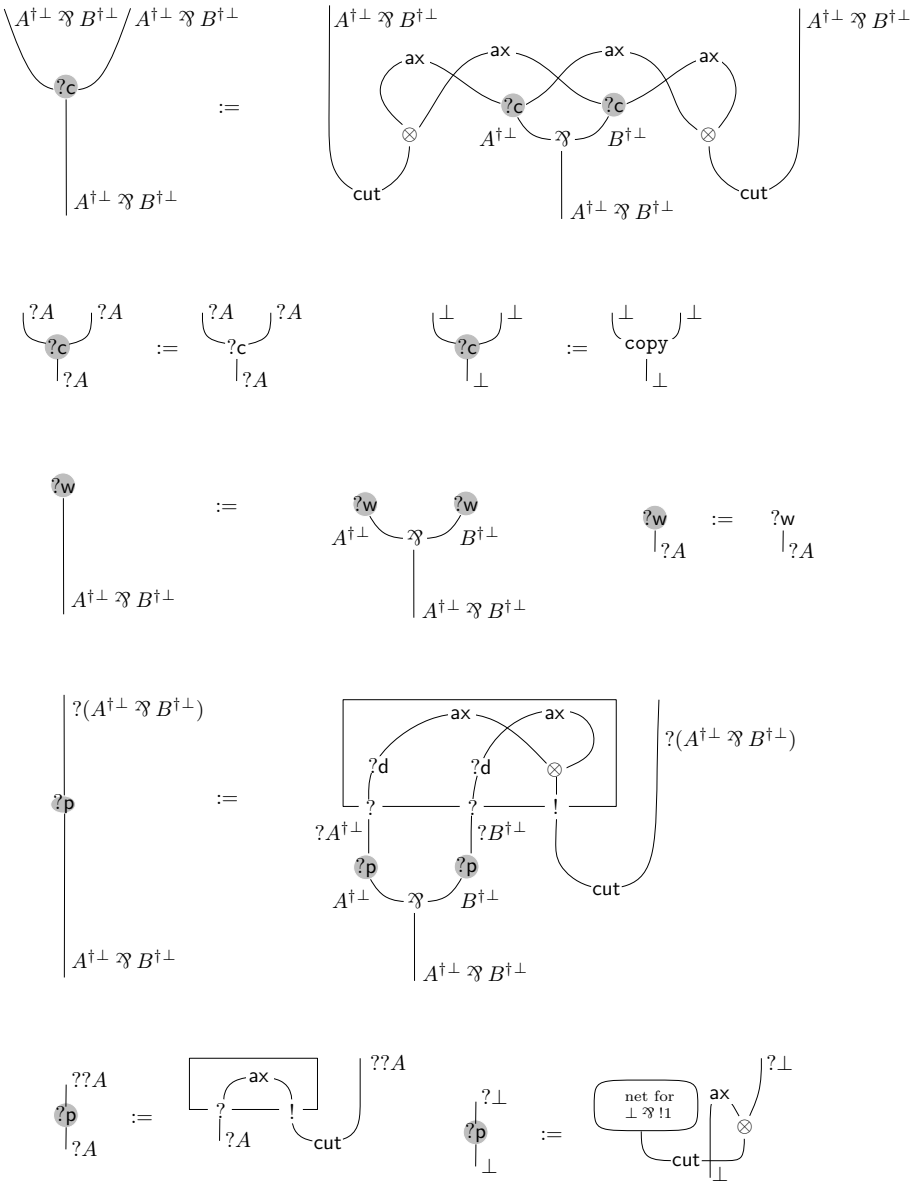


Figure 4.37: "Contraction", "Weakening", and "Dereliction" on $\bot$

Definition 4.88. The *call-by-value translation* from a PCF type A to a SMEYLL formula $A^\dagger$ is given by:

- $\mathbb{N}^\dagger := \mathsf{1}$
- $(A \to B)^\dagger := !(A^{\dagger\perp} \parr B^\dagger)$
- $(A \times B)^\dagger := A^\dagger \otimes B^\dagger$.

The *call-by-value translation* $(-)^\dagger$ from PCF type derivation trees to PCF nets is given in Figure 4.39.

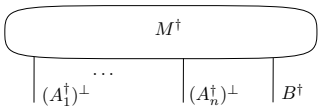


Figure 4.38: General Form of Call-by-Value Translation of PCF

For the call-by-value translation, the following statements similar to the call-by-name case all hold.

Fact 4.89. Let M be a closed term of type $\mathbb{N}$. Then its translation $\mathrm{M}^\dagger$ is a PCF net with conclusion 1.

Theorem 4.90 (Adequacy between PCF Term and PCF Net)**.** *Let* M *be a closed term of type* $\mathbb{N}$*. Then* $\mathrm{M} \Downarrow_{\mathrm{cbv}} \overline{n}$ *if and only if* $\mathrm{M}^\dagger \Downarrow n$*.*

The proof is deferred to 4.5.6.

Corollary 4.91 (Adequacy between PCF Term and PSIAM)**.** *Let* M *be a closed term of type* $\mathbb{N}$*. Then* $\mathrm{M} \Downarrow_{\mathrm{cbv}} \overline{n}$ *if and only if* $\mathcal{M}_{\mathrm{M}^\dagger} \Downarrow n$*.*

Proof. Follows from Theorem 4.76 and Theorem 4.90. □

4.5.6 Proof of Theorem 4.86 and Theorem 4.90

The chapter ends with the proof of Theorem 4.86 (Theorem 4.90 also follows in a similar way). A technical difficulty comes from the fact that one step of reduction in PCF is simulated by possibly more than one reduction in PCF nets. Especially, substitution caused by β reduction becomes problematic when we would like to look at the corresponding reductions in PCF nets. To overcome the difficulty, we introduce an intermediate language called *extPCF*. Formally, the intermediate language extPCF is defined as follows.

Definition 4.92. The syntax of terms of extPCF is the one of PCF, extended with a constructor $\mathtt{subst}\, x_1, \ldots x_n \,\mathtt{by}\, N_1, \ldots N_n \,\mathtt{in}\, M$ where $x_1, \ldots x_n$ are variables and $N_1, \ldots N_n$ are terms of extPCF. Instead of lists $x_1, \ldots, x_n$ and $N_1, \ldots, N_n$ we write $\vec{x}$ and $\vec{N}$.

The typing rule for this new construct is as follows.

$$\frac{\Delta, x_1 : A_1, \ldots x_n : A_n \vdash M : B \quad \cdot \vdash N_i : A_i \quad \text{for all } i}{\Delta \vdash \mathtt{subst}\, \vec{x} \,\mathtt{by}\, \vec{N} \,\mathtt{in}\, M : B}$$

So in particular, all substituting terms are closed.

Rewriting rules are also extended as follows. The new rewriting relation is denoted by $\to_{\mathrm{cbnext}}$.

- Call-by-name contexts are not touched (i.e. no rewriting is allowed to occur under $\mathtt{subst}\, \vec{x} \,\mathtt{by}\, \vec{N} \,\mathtt{in}\, M$).

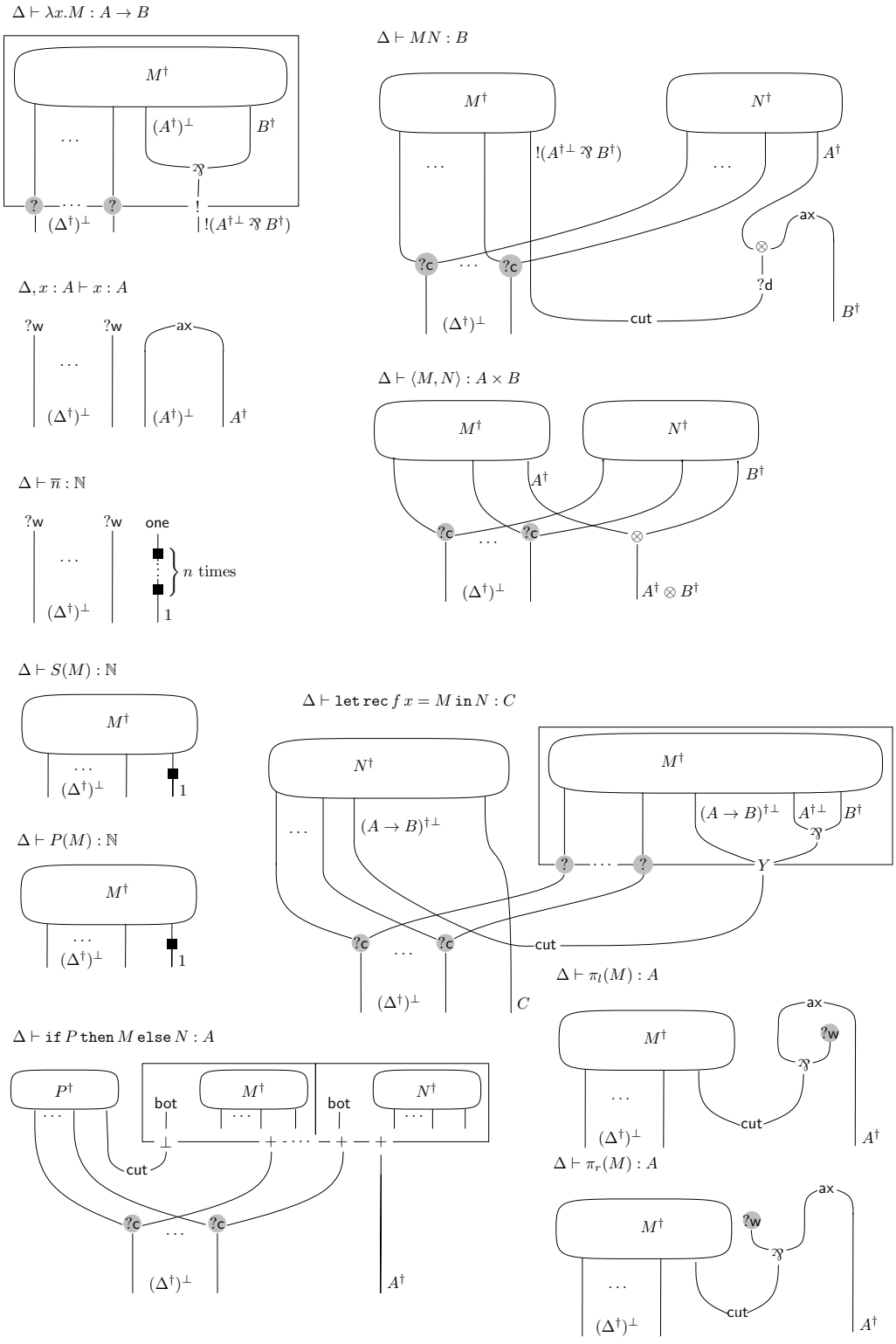


Figure 4.39: Call-by-Value Translation of PCF

- The beta redex is rewritten by

$$(\lambda x.M)N \to_{\mathrm{cbnext}} \mathtt{subst}\, x\, \mathtt{by}\, N\, \mathtt{in}\, M$$

and the let-rec rewrites by

$$\mathtt{letrec}\, f\, x = M\, \mathtt{in}\, N \to_{\mathrm{cbnext}} \mathtt{subst}\, f\, \mathtt{by}\, (\lambda x.\mathtt{letrec}\, f\, x = M\, \mathtt{in}\, f\, x)\, \mathtt{in}\, N.$$

The other axiom rules remain the same as $\to_{\mathrm{cbn}}$.

- New rewriting rules on the constructor $\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M$ are added as follows. We assume that alpha conversion is applied accordingly, and that y is not among the x_i's.

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, x_i \to_{\mathrm{cbnext}} N_i$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, y \to_{\mathrm{cbnext}} y$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, MP \to_{\mathrm{cbnext}} (\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M)(\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, P)$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \lambda y.M \to_{\mathrm{cbnext}} \lambda y.(\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M)$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \langle M, P\rangle \to_{\mathrm{cbnext}} \langle \mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M, \mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, P\rangle$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \pi_l(M) \to_{\mathrm{cbnext}} \pi_l(\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M)$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \pi_r(M) \to_{\mathrm{cbnext}} \pi_r(\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M)$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \overline{n} \to_{\mathrm{cbnext}} \overline{n}$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \mathtt{succ}(M) \to_{\mathrm{cbnext}} \mathtt{succ}(\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M)$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \mathtt{pred}(M) \to_{\mathrm{cbnext}} \mathtt{pred}(\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M)$$

$$\begin{aligned}&\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \mathtt{if}\, P\, \mathtt{then}\, M\, \mathtt{else}\, M'\\ &\to_{\mathrm{cbnext}} \mathtt{if}\, (\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, P)\, \mathtt{then}\, (\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M)\, \mathtt{else}\, (\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M')\end{aligned}$$

$$\begin{aligned}&\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \mathtt{letrec}\, f\, y = M\, \mathtt{in}\, P\\ &\to_{\mathrm{cbnext}} \mathtt{letrec}\, f\, y = (\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M)\, \mathtt{in}\, \mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, P\end{aligned}$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \mathtt{subst}\, y\, \mathtt{by}\, P\, \mathtt{in}\, M \to_{\mathrm{cbnext}} \mathtt{subst}\, \vec{x}, y\, \mathtt{by}\, \vec{N}, P\, \mathtt{in}\, M$$

We call these additional rules the *substitution rewriting*.

The translation $(\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M)^*$ is defined by wrapping each N_i^* by a !-box, and connecting M^* and those boxes via cut as shown in Figure 4.40.

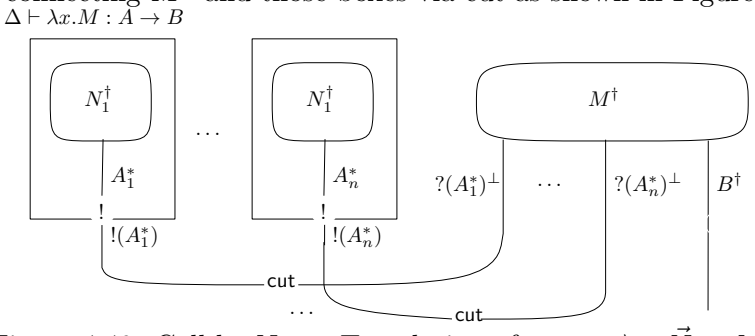


Figure 4.40: Call-by-Name Translation of $\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M$

The new constructor $\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M$ replaces the usual substitution by explicit, one-by-one substitution, which allows us to examine the correspondence between term rewriting and net reduction. In the rest of section, we prove two groups of lemmas: one for showing that extPCF is equivalent to PCF in a certain sense, and another for showing that rewriting of extPCF terms can be simulated by reduction of PCF nets translated from the terms. By using those lemmas we finally verify the theorem we aim at. First we define the following "read back" from extPCF terms to PCF terms.

Definition 4.93. A map $\downarrow$ from extPCF terms to PCF terms is recursively defined as follows:

$$\begin{aligned}
&\downarrow(x) = x\\
&\downarrow(\overline{n}) = \overline{n}\\
&\downarrow(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M) = (\downarrow M)\{x_i := \downarrow N_i \mid i = 1 \cdots n\}.\\
&\downarrow(\texttt{subst}\,y\,\texttt{by}\,M\,\texttt{in}\,P) = \texttt{subst}\,y\,\texttt{by}\downarrow(M)\,\texttt{in}\downarrow(P)\\
&\downarrow(\texttt{succ}(M)) = \texttt{succ}(\downarrow(M))\\
&\downarrow(\texttt{pred}(M)) = \texttt{pred}(\downarrow(M))\\
&\downarrow(\mathrm{MN}) = \downarrow(\mathrm{M})\downarrow(\mathrm{N})\\
&\downarrow(\lambda y.M) = \lambda y.\downarrow(M)\\
&\downarrow(\langle M, N\rangle) = \langle\downarrow(M), \downarrow(N)\rangle\\
&\downarrow(\pi_l M) = \pi_l\downarrow(M)\\
&\downarrow(\pi_r M) = \pi_r\downarrow(M)\\
&\downarrow(\texttt{if}\,P\,\texttt{then}\,M\,\texttt{else}\,N) = \texttt{if}\downarrow(P)\,\texttt{then}\downarrow(M)\,\texttt{else}\downarrow(N)\\
&\downarrow(\texttt{letrec}\,f\,y = M\,\texttt{in}\,P) = \texttt{letrec}\,f\,y = \downarrow(M)\,\texttt{in}\downarrow(P).
\end{aligned}$$

Lemma 4.94. *Let M be an extPCF term such that $M \to_{\mathrm{cbnext}} N$. If it is by a substitution rewriting, then $\downarrow M = \downarrow N$ holds. Otherwise $\downarrow M \to^{+}_{\mathrm{cbn}} \downarrow N$ holds.*

Proof. By structural induction on the derivation of $M \to_{\mathrm{cbnext}} N$. □

Lemma 4.95. *Let M be a PCF term. If $M \to^{*}_{\mathrm{cbnext}} \overline{n}$, then $M \to^{*}_{\mathrm{cbn}} \overline{n}$ holds.*

Proof. Observe that, since M is a PCF-term, M does not contain any term in the form of $\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M$, and thus $\downarrow M = M$ by definition. Observe also that $\downarrow\overline{n} = \overline{n}$ by definition. Then the lemma follows by induction on the length of the rewriting sequence $M \to^{*}_{\mathrm{cbnext}} \overline{n}$, invoking Lemma 4.94 in the induction case. □

Lemma 4.96. *Let P be a term in extPCF such that $\downarrow P \to_{\mathrm{cbn}} M$. Then there exists a term P' in extPCF such that $\downarrow P' = M$ and $P \to^{+}_{\mathrm{cbnext}} P'$.*

Proof. Proof by induction on the size of P. □

Lemma 4.97. *Suppose that N does not substitution-reduce. Suppose furthermore that $\downarrow M = \downarrow N$, and that $M \to_{\mathrm{cbnext}} M'$ is not a substitution-reduction step. Then there exists N' such that $\downarrow M' = \downarrow N'$ and $N \to^{+}_{\mathrm{cbnext}} N'$.*

Proof. The proof is done by structural induction on M, and by case distinction on $M \to_{\mathrm{cbnext}} M'$. □

Definition 4.98. We define the substitution-size $ss(M)$ of a term M inductively as follows.

- $ss(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M)$ is defined as

$$\left(\prod_i (1 + ss(N_i))\right)^{ss(M)}$$

- the substitution-size of terms built from any other constructors is 1 plus the sum of the substitution size of their constituents. For example:

$$ss(x) = ss(\overline{n}) = 1$$

$$ss(\pi_l M) = 1 + ss(M)$$

$$ss(MN) = ss(\langle M, N\rangle) = 1 + ss(M) + ss(N)$$

$$ss(\texttt{letrec}\,f\,x = M\,\texttt{in}\,N) = 1 + ss(M) + ss(N)$$

$$ss(\texttt{if}\,P\,\texttt{then}\,M\,\texttt{else}\,N) = 1 + ss(M) + ss(N) + ss(P)$$

Lemma 4.99. *If P is a strict subterm of M, then $ss(P) < ss(M)$. If M substitution-reduces to N, then $ss(N) < ss(M)$.*

Proof. The first part of the lemma is easy to check by structural induction on M, realizing that for all M, $ss(M) \geq 1$. The second part of the lemma is shown by induction on the derivation of $M \to_{\text{cbnext}} N$.

- $\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,MP \to_{\text{cbnext}} (\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M)(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,P)$.

  Provided that we set $x := \prod_i (1 + ss(N_i))$, we have to show

  $$x^{1+ss(M)+ss(P)} > 1 + x^{ss(M)} + x^{ss(P)}.$$

  With $a := x^{ss(M)}$ and $b := x^{ss(P)}$, this can be rewritten as

  $$x \cdot a \cdot b > 1 + a + b.$$

  Since $x \geq 2$, it is enough to show that

  $$2 \cdot a \cdot b > 1 + a + b.$$

  This is equivalent to

  $$a \cdot (2 \cdot b - 1) > b + 1.$$

  Since $a \geq 2$, it is enough to show that

  $$2 \cdot (2 \cdot b - 1) > b + 1,$$

  that is,

  $$3 \cdot b - 3 > 0.$$

  Since $b \geq 2$, this is always verified.

- $\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,\lambda y.M \to_{\text{cbnext}} \lambda y.(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M)$.

  Provided that we set $x := \prod_i (1 + ss(N_i))$, we have to show

  $$x^{1+ss(M)} > 1 + x^{ss(M)}.$$

  With $a := x^{ss(M)}$, and since $x \geq 2$, it is enough to show

  $$2 \cdot a > 1 + a.$$

  This inequality is valid since $a \geq 2$.

- $\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,\texttt{if}\,P\,\texttt{then}\,M\,\texttt{else}\,M' \to_{\text{cbnext}}$
  $\texttt{if}\,(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,P)\,\texttt{then}\,(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M)\,\texttt{else}\,(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M)$.

  Provided that we set $x := \prod_i (1 + ss(N_i))$, we have to show

  $$x^{1+ss(P)+ss(M)+ss(M')} > 1 + x^{ss(P)} + x^{ss(M)} + x^{ss(M')}.$$

  With $a := x^{ss(P)}$, $b := x^{ss(M)}$ and $c := x^{ss(M')}$ this can be rewritten as

  $$x \cdot a \cdot b \cdot c > 1 + a + b + c.$$

  Since $x \geq 2$, it is enough to show that

  $$2 \cdot a \cdot b \cdot c > 1 + a + b + c.$$

  Since b and c are larger or equal to 2, we have $b \cdot c > b + c$, and writing $d := b + c$, it is enough to show

  $$a \cdot (2 \cdot d - 1) > d + 1.$$

  We are back to the situation of first bullet point: this inequality is then valid.

- $\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,\texttt{subst}\,\vec{y}\,\texttt{by}\,\vec{M}\,\texttt{in}\,P \to_{\text{cbnext}} \texttt{subst}\,\vec{x},\vec{y}\,\texttt{by}\,\vec{N},\vec{M}\,\texttt{in}\,P$.

  Provided that we set $x := \prod_i (1 + ss(N_i))$, $y := \prod_i (1 + ss(M_i))$ and that we write $a := ss(P)$, we have to show

  $$x^{y^a} > (x \cdot y)^a.$$

  Since $y \geq 2$ and $a \geq 1$, we have $y^a > y \cdot a$. It is then enough to show

  $$x^{y \cdot a} > (x \cdot y)^a.$$

  This is equivalent to

  $$(x^y)^a > (x \cdot y)^a,$$

  and it is correct since $x^y > x \cdot y$.

The other cases are treated similarly. □

Lemma 4.100. *Suppose that $\downarrow M = \downarrow N$, and that $M \to^*_{\text{cbnext}} b$, where b is either a term variable or a constant $\overline{n}$. Then $N \to^*_{\text{cbnext}} b$.*

Proof. The proof is done by induction on the substitution-size of N.

- Base case: $N = c$, where c is a term constant or a term variable. We need to show that $c = b$.

  We proceed by induction on the size of the sequence of reduction from M to b.

  - If $M = b$, then clearly $c = b$.
  - Suppose that for all terms reducing in n steps to b then $c = b$. Consider a sequence of reduction $M \to_{\text{cbnext}} M' \to^*_{\text{cbnext}} b$ of size $n + 1$. Since $N = c$, we have $\downarrow M = c$. There are two cases:
    - Either $M = c$, in which case we get a contradiction since M cannot reduce to any M'.

* Or $M = \texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{P}\,\texttt{in}\,P'$. In this case, M' is necessarily coming from a substitution-reduction, meaning that $\downarrow M' = \downarrow M = c$. By induction hypothesis we conclude that $c = b$.

Therefore, in the case where N is a constant or a variable, it is indeed equal to b.

- Now, suppose that it is neither a constant nor a variable, and that the result is correct for all terms of smaller substitution-size.

  We again proceed by induction on the size of the sequence of reduction from M to b.

  - If $M = b$, then $\downarrow N = b$. There are two cases.
    * Either $N = b$, and we are done.
    * Or $N = \texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{P}\,\texttt{in}\,P'$. It then reduces through a substitution-reduction to some term that is substitution-smaller than N, from Lemma 4.99. We can then invoke the induction hypothesis, and deduce that N reduces to b.
  - Otherwise, suppose that $M \to_{\text{cbnext}} M' \to^*_{\text{cbnext}} b$, and that the result is true for M'. We know that $\downarrow M = \downarrow M'$. Without loss of generality one can suppose that N does not substitution-reduce: otherwise, we could invoke Lemma 4.99 and the outer-most induction hypothesis as above to conclude.

    We proceed by case distinction on $M \to_{\text{cbnext}} M'$.
    * If it is a substitution-reduction step, then $\downarrow M = \downarrow M' = \downarrow N$, and the inner induction hypothesis tells us that $N' \to^*_{\text{cbnext}} b$.
    * Otherwise, we first apply Lemma 4.97, conclude to the existence of N' such that $N \to_{\text{cbnext}} N'$ with $\downarrow N' = \downarrow M'$, and apply the inner induction hypothesis to conclude.

This closes the proof of the lemma. □

Lemma 4.101. *If N is a term in extPCF such that $\downarrow N = b$, where b is either a term variable or a constant $\overline{n}$, then $N \to^*_{\text{cbnext}} b$.*

Proof. This is a corollary of Lemma 4.100, when setting $M = b$. Indeed, in that case we trivially have that $\downarrow M = \downarrow N$ and that $M \to^*_{\text{cbnext}} b$, so the lemma applies. □

Lemma 4.102. *If M is a PCF term, then $M \to^*_{\text{cbn}} \overline{n}$ if and only if $M \to^*_{\text{cbnext}} \overline{n}$.*

Proof. The right-to-left direction is Lemma 4.95. For the left-to-right direction, remark that $\downarrow M = M$ and $\downarrow \overline{n} = \overline{n}$. Then consider the reduction sequence

$$M \to_{\text{cbn}} M_1 \to_{\text{cbn}} \cdots \to_{\text{cbn}} M_n = \overline{n}.$$

Applying Lemma 4.96 on each step of this sequence, one construct a sequence of reduction showing that $M \to^*_{\text{cbnext}} M'_n$ where $\downarrow M'_n = \overline{n}$. We finally conclude with Lemma 4.101 □

Lemma 4.103. *Suppose that M and N are closed terms of type A. If $M \to_{\text{cbnext}} N$, then $M^* \to^+ N^*$.*

Proof. The proof is done by structural induction on the derivation of the reduction $M \to_{\text{cbnext}} N$. □

Lemma 4.104. *Suppose that a net R converges to a normal form. Then any sequence of reduction starting from R is finite and terminates on the same normal form.*

Proof. As in Proposition 4.9. □

Lemma 4.105. *Suppose that M is a closed term of type $\mathbb{N}$. Then $M \to^*_{\text{cbnext}} \overline{n}$ if and only if $M^* \to^* \overline{n}^*$.*

Proof. The left-to-right direction is proven by simple induction on the size of the rewrite sequence $M \to^*_{\text{cbnext}} \overline{n}$, using Lemma 4.103.

For the right-to-left direction, suppose that $M^* \to^* \overline{n}^*$ but that there exists an infinite sequence $\{M_i\}_{i\in\mathbb{N}}$ such that $M = M_0$ and such that for all i, $M_i \to_{\text{cbnext}} M_{i+1}$. Then using Lemma 4.103 we can conclude that there is an infinite net-rewrite sequence starting with M^*. From Lemma 4.104 this contradicts the fact that M^* converges. □

Proof of Theorem 4.86. The desired adequacy result simply follows from the use of Lemma 4.102 to fall back on the intermediate PCF and Lemma 4.105.

Chapter 5

Memory-Based Synchronous Interaction Abstract Machine

In the previous chapter, we saw how our multi-token framework can successfully interpret a PCF-like programming language in a uniform manner. Since the initial motivation for the multi-token framework came from quantum computation, it is now natural to ask: then, is it able to adequately interpret some quantum programming language using the same framework? In the chapter, we are going to answer the question, as expected positively. Moreover, what we can state is more general. We define a class of proof net systems, multi-token machine systems, and PCF-like languages, each parameterized by a notion of *memory structure*. The deterministic language we studied in §4.5, a quantum language (with exponentials and recursion) akin to the one studied in [79], and a probabilistic language akin to the one in [26] are all shown to be certain *instances* of the parametrically defined language. Finally we show an adequacy result also in a parameterized way, thus the adequacy result for each language is automatically obtained as soon as we fix a specific memory structure.

Besides those results on interpretation of languages, we use the notion of *probabilistic abstract rewriting systems (PARSs)* to define and analyze the probabilistic systems. It is also of independent interest since the topic seems undeveloped as far as the author (and coauthors of the paper [19]) know. Especially, the definition of (an analogue of) the diamond property and providing a sufficient condition for the property of PARS is novel.

The content of this chapter is based on a published paper [19]. Some part of proofs and formulation of the notion of memory structure is by Claudia Faggian and Benoît Valiron. The initial idea of PARSs is due to Ugo Dal Lago.

Organization of the chapter. In Section 5.1 we introduce the notion of probabilistic abstract rewriting systems (PARSs) and show some basic, important properties in PARSs. Section 5.2 contains the notion of memory structures, with some concrete instances of memory structures. We then equip proof nets, token machines, and a language uniformly with the notion of memory structures in the following sections (Section 5.3, 5.4, 5.5). As in the classical case translations from the language to proof nets are defined, and soundness and adequacy will be shown parametrically on memory structures.

5.1 Probabilistic Abstract Rewriting System

In the chapter we introduce a class of proof net systems, a class of multi-token machines, and a class of programming languages. All of them are defined as

probabilistic abstract rewriting systems (PARSs) that we define and investigate in the section.

Definition 5.1 (Probabilistic Abstract Rewriting System)**.** A *probabilistic abstract rewriting system* $\mathbf{A}$ is a pair $(\mathrm{A}, \to)$ where A is a set and $\to$ is a relation $\to \subseteq \mathrm{A} \times \mathrm{Dist}(\mathrm{A})$ satisfying that: if $(a, \mu) \in \to$ then $\mathrm{supp}(\mu)$ is finite for all $a \in \mathrm{A}$ and $\mu \in \mathrm{Dist}(\mathrm{A})$. (See Section 3.1 for the definition of $\mathrm{Dist}(\mathrm{A})$)

Definition 5.2 (Partition of Distribution)**.** Given a probabilistic abstract rewriting system $\mathbf{A} = (\mathrm{A}, \to)$ and a distribution $\mu \in \mathrm{Dist}(\mathrm{A})$, two distributions $\mu^\circ, \bar{\mu} \in \mathrm{Dist}(\mathrm{A})$ are defined as follows:

$$\mu^\circ(a) = \begin{cases} \mu(a) & \text{if } a \not\to, \\ 0 & \text{otherwise;} \end{cases} \qquad \bar{\mu}(a) = \mu(a) - \mu^\circ(a).$$

Definition 5.3 (Degree of Termination)**.** Given a probabilistic abstract rewriting system $\mathbf{A} = (\mathrm{A}, \to)$ and a distribution $\mu \subseteq \mathrm{Dist}(\mathrm{A})$, the *degree of termination* of μ is defined as $\mathcal{T}(\mu) = \sum_{a \in \mathrm{A}} \mu^\circ(a)$.

As clear from the definition, μ° and $\bar{\mu}$ partition the distribution μ into the distribution of the elements that are in normal form with respect to the relation $\to$, and the other distribution of the elements that can still evolve by $\to$. The degree of termination $\mathcal{T}(\mu)$ is thus the total probability of terminated elements in the distribution μ.

Unlike ARSs, the types of the "domain" and the "codomain" of $\to$ differ, thus it is not trivial to define a transitive closure of $\to$ for a PARS. Here we define a rather restricted relation $\rightrightarrows \subseteq \mathrm{Dist}(\mathrm{A}) \times \mathrm{Dist}(\mathrm{A})$ between $\mathrm{Dist}(\mathrm{A})$ that *simultaneously* reduces each reducible element in a distribution. We also define another relation $\twoheadrightarrow \subseteq \mathrm{A} \times \mathrm{Dist}(\mathrm{A})$ that reduces elements *one by one*. We use the latter later in the chapter.

Definition 5.4 (Relation $\rightrightarrows$ and $\twoheadrightarrow$)**.** Let $\mathbf{A} = (\mathrm{A}, \to)$ be a probabilistic abstract rewriting system. The relation $\rightrightarrows \subseteq \mathrm{Dist}(\mathrm{A}) \times \mathrm{Dist}(\mathrm{A})$ is defined by the following rule.

$$\frac{\mu = \mu^\circ + \bar{\mu} \quad \{a \to \nu_a\}_{a \in \mathrm{supp}(\bar{\mu})}}{\mu \rightrightarrows \mu^\circ + \sum_{a \in \mathrm{supp}(\bar{\mu})} \mu(a) \cdot \nu_a}\ .$$

The relation $\twoheadrightarrow \subseteq \mathrm{Dist}(\mathrm{A}) \times \mathrm{Dist}(\mathrm{A})$ is defined by the following rule.

$$\frac{a \to \mu}{a \twoheadrightarrow \mu} \qquad \frac{}{a \twoheadrightarrow \{a^1\}} \qquad \frac{a \twoheadrightarrow \mu + \{b^p\} \quad b \twoheadrightarrow \rho \quad b \notin \mathrm{supp}(\mu)}{a \twoheadrightarrow \mu + p \cdot \rho}$$

Remark 5.5. Even if we regard an element $a \in \mathrm{A}$ in a PARS $(\mathrm{A}, \to)$ as a distribution $\{a^1\}$, the distributions reachable by $\rightrightarrows$ and $\twoheadrightarrow$ differs in general. As an example, consider a PARS $(\{i, a, b, c, d, e\}, \to)$ defined by $i \to \{a^{1/2}, b^{1/2}\}$, $a \to \{c^1\}$, $b \to \{c^1\}$, $c \to \{d^1\}$, and $c \to \{e^1\}$. In this example $i \twoheadrightarrow \{d^{1/2}, e^{1/2}\}$ holds (for example, first reduce a in $\{a^{1/2}, b^{1/2}\}$ to d, and then b to e). The distribution is unreachable by $\rightrightarrows$, that is, $\{i^1\} \rightrightarrows \{d^{1/2}, e^{1/2}\}$ does not hold, although $\{a^{1/2}\} \rightrightarrows \{d^{1/2}\}$ and $\{b^{1/2}\} \rightrightarrows \{e^{1/2}\}$ hold. This is because $\{i^1\} \rightrightarrows \{a^{1/2}, b^{1/2}\} \rightrightarrows \{c^1\}$ and either $\{c^1\} \rightrightarrows \{d^1\}$ or $\{c^1\} \rightrightarrows \{e^1\}$ only holds.

Remark 5.6. The definition of PARS itself is essentially the same as Markov decision process. The use of the relation $\rightrightarrows$ is however unusual in the study of Markov decision processes, and plays an essential role to make it possible to define notions analogous to abstract reduction systems.

Based on the definition of $\rightrightarrows$ above, we define notions corresponding to *normalization*, *convergence*, and *confluence* of ARSs. The first two inevitably become quantitative, while confluence is defined to be a usual one with respect to the relation $\rightrightarrows$.

Definition 5.7 (p-Normalization)**.** For a distribution μ,

- μ *weakly p-normalizes* if there exists ν satisfying $\mu \rightrightarrows^* \nu$ and $\mathcal{T}(\nu) \geq p$.
- μ *strongly p-normalizes* (or *p-terminates*) if there exists $n \in \mathbb{N}$ such that $\mu \rightrightarrows^n \nu$ implies $\mathcal{T}(\nu) \geq p$ for any such ν.

Definition 5.8 (Convergence with Probability p)**.** Given a probabilistic abstract rewriting system $(\mathrm{A}, \to)$, a distribution $\mu \in \mathrm{Dist}(\mathrm{A})$ *converges with probability p*, denoted by $\mu\Downarrow_p$, if $p = \sup_{\mu \rightrightarrows^* \nu} \mathcal{T}(\nu)$.

Definition 5.9 (Confluence)**.** A probabilistic abstract rewriting system $(\mathrm{A}, \to)$ is said to be *confluent* if $\mu \rightrightarrows^* \nu_1$ and $\mu \rightrightarrows^* \nu_2$ implies that there exists ξ satisfying $\nu_1 \rightrightarrows^* \xi$ and $\nu_2 \rightrightarrows^* \xi$.

On top of the notions defined above, we define an important property analogous to the diamond property of ARSs. The first condition in the definition below is specific to our probabilistic setting. It is to ensure Theorem 5.12.

Definition 5.10 (Diamond Property of PARS)**.** A probabilistic abstract rewriting system $(\mathrm{A}, \to)$ is said to satisfy the *diamond property* if the following holds: if $\mu \rightrightarrows \nu_1$ and $\mu \rightrightarrows \nu_2$, then

1. ${\nu_1}^\circ = {\nu_2}^\circ$,
2. and there exists $\xi \in \mathrm{Dist}(\mathrm{A})$ satisfying $\nu_1 \rightrightarrows \xi$ and $\nu_2 \rightrightarrows \xi$.

Corollary 5.11. *If a probabilistic abstract rewriting system $(\mathrm{A}, \to)$ satisfies the diamond property, then $(\mathrm{A}, \to)$ is confluent.*

Proof. Immediate by definition. □

Finally, we state and prove a property that we heavily use later. On the one hand, it assures that weak p-normalization implies strong p-normalization under the diamond property, which is alike what often stated for ARSs. On the other hand, the property we call *uniqueness of normal forms* has no counterpart in ARSs and allows us to prove some properties we want later in the chapter.

Theorem 5.12 (Uniqueness of Normal Forms and Uniformity)**.** *Assume a probabilistic abstract rewriting system $(\mathrm{A}, \to)$ satisfies the diamond property. Then we have the following properties.*

1. ***Uniqueness of normal forms.*** *For any $k \in \mathbb{N}$ and $\mu \in \mathrm{Dist}(\mathrm{A})$, $\mu \rightrightarrows^k \nu$ and $\mu \rightrightarrows^k \xi$ implies $\nu^\circ = \xi^\circ$.*
2. ***Uniformity.*** *If μ is weakly p-normalizing for some $p \in [0, 1]$, then μ is strongly p-normalizing.*

Proof. Item 2. follows from 1. We prove item 1. by an adaptation of the tiling argument used for usual ARSs. It is not exactly the same as the standard proof, because even if an element in a distribution μ is in normal form there can still be non-normal ones in μ, and thus μ can be reduced by $\rightrightarrows$.

Assume $\mu = \nu_0 \rightrightarrows \nu_1 \rightrightarrows \cdots \rightrightarrows \nu_k$ and $\mu \rightrightarrows \xi_1 \rightrightarrows \cdots \rightrightarrows \xi_k$ for some $k \in \mathbb{N}$. We prove ${\nu_k}^\circ = {\xi_k}^\circ$ by induction on k. If $k = 1$ it holds by the diamond property (Definition 5.10.1). If $k > 1$, we make a tiling (w.r.t. $\rightrightarrows$) as depicted in Figure 5.1: we build the sequence $\xi_1 = \rho_0 \rightrightarrows \rho_1 \rightrightarrows \cdots \rightrightarrows \rho_{k-1}$ (see Figure 5.1) where each ρ_{i+1} $(i \geq 0)$ is obtained by the diamond property (Definition 5.10.2), applied for $\nu_i \rightrightarrows \nu_{i+1}$ and $\nu_i \rightrightarrows \rho_i$. By Definition 5.10.1, ${\nu_k}^\circ = {\rho_{k-1}}^\circ$. We have $\xi_1 \rightrightarrows^{k-1} \xi_k$

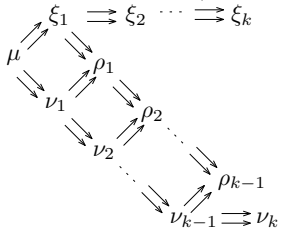


Figure 5.1: Tiling in PARS

and $\xi_1 \rightrightarrows^{k-1} \rho_{k-1}$, thus by induction hypothesis ${\xi_k}^\circ = {\rho_{k-1}}^\circ$. Hence we conclude that ${\nu_k}^\circ = {\xi_k}^\circ$. □

Remark 5.13. The diamond property is a rather strong property, and it is true that not so many PARSs satisfy it. Although it would be interesting to search for more moderate properties and investigate the general theory of PARSs further, we stop here because the properties we introduced so far are at least sufficient for our purpose in the thesis.

5.2 Memory Structures

In section 4.5, we interpreted the PCF-like language by SMEYLL nets and the SIAM extended with a notion of memory consisting of natural numbers. The content of a memory should be different if we consider another language with a different data type, such as integers, real numbers, or quantum states. In the section we introduce a generalized notion to deal with such various memories, called *memory structure*, with a requirement that the operations on memories are commutative in a certain sense. The generalized notion allows us to *parameterize* the notions of proof nets, multi-token machines, and programming languages, hence leading to a unified handling of different kinds of languages in one single framework. The notion of memory structure is defined to be a certain kind of nominal sets; this is for convenience rather than a theoretical insight.

Definition 5.14 (Memory Structure)**.** A *memory structure* is a 4-tuple $(\mathrm{Mem}, \cdot, \mathcal{I}, \mathcal{L})$ where

- $(\mathrm{Mem}, \cdot)$ is a nominal set,
- $\mathcal{I}$ is a countably infinite set whose elements are called *indexes*,
- $\mathcal{L}$ is a finite set whose elements are called *operations*,

equipped with the following maps

- $\mathrm{test} \colon \mathcal{I} \times \mathrm{Mem} \to \mathrm{Dist}(\mathbb{B}) \times (\mathbb{B} \to \mathrm{Mem})$
- $\mathrm{update} \colon \mathcal{I}^* \times \mathcal{L} \times \mathrm{Mem} \rightharpoonup \mathrm{Mem}$
- $\mathrm{arity} \colon \mathcal{L} \to \mathbb{N}$

satisfying the conditions listed below. The type $\mathrm{Dist}(\mathbb{B}) \times (\mathbb{B} \to \mathrm{Mem})$ that the function test returns is meant to represent distributions in the form of $\{(\texttt{true}, \mathrm{m}_0)^{p_0}, (\texttt{false}, \mathrm{m}_1)^{p_1}\}$, and we write $\mathrm{test}(i, \mathrm{m}) = \{(\texttt{true}, \mathrm{m}_0)^{p_0}, (\texttt{false}, \mathrm{m}_1)^{p_1}\}$ for $\mathrm{test}(i, \mathrm{m}) = (\{\texttt{true}^{p_0}, \texttt{false}^{p_1}\}, \{\texttt{true} \mapsto \mathrm{m}_0, \texttt{false} \mapsto \mathrm{m}_1\})$.

1. The maps test and update respect the group action $\cdot$ of the nominal set $(\mathrm{Mem}, \cdot)$. Explicitly,

$$\sigma \cdot (\mathrm{test}(i, \mathrm{m})) = \mathrm{test}(\sigma(i), \sigma \cdot \mathrm{m}),$$
$$\sigma \cdot (\mathrm{update}(\vec{i}, x, \mathrm{m})) = \mathrm{update}(\sigma(\vec{i}), x, \sigma \cdot \mathrm{m}).$$

2. The map update is defined if and only if the arity of a given operation and the number of given distinct indexes coincide. Explicitly, $\mathrm{update}((i_1, \ldots, i_n), \ell, \mathrm{m})$ is defined if and only if the indexes i_k's are pairwise disjoint and $\mathrm{arity}(\ell) = n$.

3. The maps test and update commute if they act on different indexes. Explicitly,

   - Tests on i commute with tests on j. More precisely, if
     - $\mathrm{test}(i, \mathrm{m}) = \{(\texttt{true}, \mathrm{m}_0)^{p_0}, (\texttt{false}, \mathrm{m}_1)^{p_1}\}$
     - $\mathrm{test}(j, \mathrm{m}_0) = \{(\texttt{true}, \mathrm{m}_{00})^{p_{00}}, (\texttt{false}, \mathrm{m}_{01})^{p_{01}}\}$
     - $\mathrm{test}(j, \mathrm{m}_1) = \{(\texttt{true}, \mathrm{m}_{10})^{p_{10}}, (\texttt{false}, \mathrm{m}_{11})^{p_{11}}\}$

     and
     - $\mathrm{test}(j, \mathrm{m}) = \{(\texttt{true}, \mathrm{m}'_0)^{q_0}, (\texttt{false}, \mathrm{m}'_1)^{q_1}\}$
     - $\mathrm{test}(i, \mathrm{m}'_0) = \{(\texttt{true}, \mathrm{m}'_{00})^{q_{00}}, (\texttt{false}, \mathrm{m}'_{01})^{q_{01}}\}$
     - $\mathrm{test}(i, \mathrm{m}'_1) = \{(\texttt{true}, \mathrm{m}'_{10})^{q_{10}}, (\texttt{false}, \mathrm{m}'_{11})^{q_{11}}\}$

     then for all $x, y \in \{0, 1\}$, $\mathrm{m}_{xy} = \mathrm{m}'_{yx}$ and $p_x p_{xy} = q_y q_{yx}$.
   - Tests of j commute with updates on $\vec{k}$. More precisely, if
     - $\mathrm{test}(i, \mathrm{m}) = \{(\texttt{true}, \mathrm{m}_0)^{p_0}, (\texttt{false}, \mathrm{m}_1)^{p_1}\}$
     - $\mathrm{update}(\vec{k}, x, \mathrm{m}_0) = \mathrm{m}'_0$
     - $\mathrm{update}(\vec{k}, x, \mathrm{m}_1) = \mathrm{m}'_1$

     and if $\mathrm{update}(\vec{k}, x, \mathrm{m}) = \mathrm{m}'$ then

$$\mathrm{test}(i, \mathrm{m}') = \{(\texttt{true}, \mathrm{m}'_0)^{p_0}, (\texttt{false}, \mathrm{m}'_1)^{p_1}\}.$$

   - Updates on $\vec{k}$ and $\vec{k}'$ commute. More precisely:

$$\mathrm{update}(\vec{k}, x, \mathrm{update}(\vec{k}', x', \mathrm{m})) = \mathrm{update}(\vec{k}', x', \mathrm{update}(\vec{k}, x, \mathrm{m})).$$

The commutation of two tests (a) and the commutation of test and update (b) can be concisely shown as Figure 5.2.

In the next three subsections, we see three typical instances of the memory structures. Note that they are just instances; whenever the conditions described in Definition 5.14 are satisfied, it is an instance of memory structures and all the parameterized arguments in Section 5.3, 5.4, and 5.5 apply.

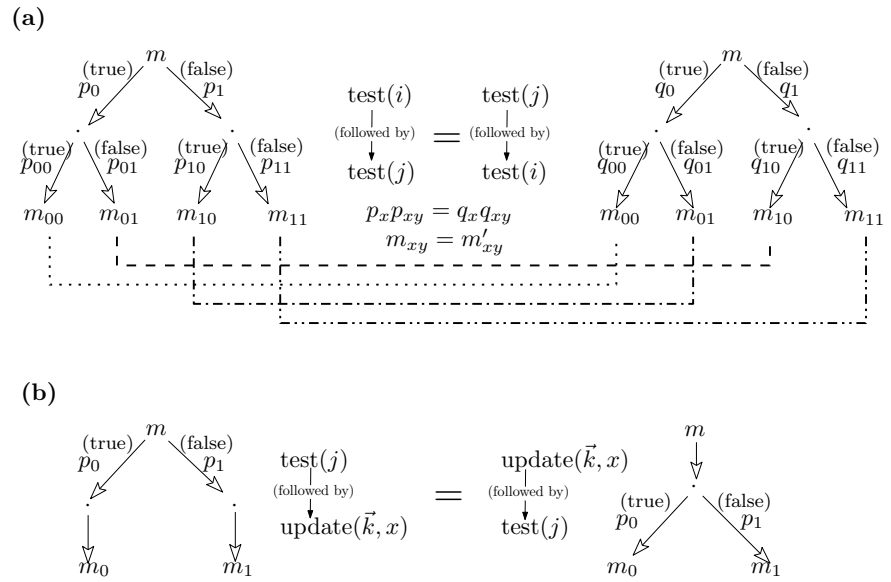


Figure 5.2: Commutation of test and update

5.2.1 Instance: Deterministic, Natural Number Memory

The first case is nothing but the memory structure used in Section 4.5. A *natural number memory* is an element m of the memory structure $\mathrm{Mem}_{\mathbb{N}} = (\mathrm{Mem}_{\mathbb{N}}, \cdot, \mathcal{I}, \mathcal{L})$ defined by:

- $\mathcal{I} = \mathbb{N}$,
- $\mathrm{Mem}_{\mathbb{N}} = \{\mathrm{m} \colon \mathcal{I} \to \mathbb{N}\}$ with $\mathrm{supp}(\mathrm{m}) = \{i \in \mathcal{I} \mid \mathrm{m}(i) \neq 0\}$,
- $(\sigma \cdot \mathrm{m})(i) := \mathrm{m}(\sigma(i))$,
- $\mathcal{L} = \{\mathtt{max}, \mathtt{succ}, \mathtt{pred}\}$, with $\mathrm{arity}(\mathtt{max}) = 2$ and $\mathrm{arity}(\mathtt{succ}) = \mathrm{arity}(\mathtt{pred}) = 1$,
- update is defined as follows.
  - $\mathrm{update}(i, j, \mathtt{max}, \mathrm{m}) = \mathrm{m}[i \mapsto \max(\mathrm{m}(i), \mathrm{m}(j)), j \mapsto \max(\mathrm{m}(i), \mathrm{m}(j))]$
  - $\mathrm{update}(i, \mathtt{succ}, \mathrm{m}) = \mathrm{m}\{i{:=}\mathrm{m}(i) + 1\}$
  - $\mathrm{update}(i, \mathtt{pred}, \mathrm{m}) = \mathrm{m}\{i{:=}\max(\mathrm{m}(i) - 1, 0)\}$
- $\mathrm{test}(i, \mathrm{m}) = \begin{cases} \{(\mathtt{true}, \mathrm{m})^1\} & \text{if } \mathrm{m}(i) = 0 \\ \{(\mathtt{false}, \mathrm{m})^1\} & \text{if } \mathrm{m}(i) > 0 \end{cases}$

We write a natural number memory m as an infinite sequence of integers $(\mathrm{m}(0), \mathrm{m}(1), \mathrm{m}(2), \ldots)$. A memory with an empty support is written $\mathrm{m}_0 = (0, 0, 0, \ldots)$ for example. If we apply the operation succ on the address 0 of the memory m_0, it rewrites into $\mathrm{m}_1 = \mathrm{update}(0, \mathtt{succ}, \mathrm{m}_0) = (1, 0, 0, 0, \ldots)$. Likewise, if we apply succ on the address 1 on m_1, $\mathrm{m}_2 = \mathrm{update}(1, \mathtt{succ}, \mathrm{m}_1) = (1, 1, 0, 0, \ldots)$, and then if we apply the operation pred on the address 0, we obtain $\mathrm{m}_3 = \mathrm{update}(0, \mathtt{pred}, \mathrm{m}_2) = (0, 1, 0, 0, \ldots)$. Applying the test operation yields $\mathrm{test}(1, \mathrm{m}_3) = \{(\mathtt{false}, \mathrm{m}_3)^1\}$, without changing any value on the memory. Note that we only need to keep track of finitely many values since having a finite support is imposed on a memory.

Remark 5.15. The equations required for memory structures enforce every memory to contain the same values in all the fresh addresses (i.e., not in the support of the memory as a nominal set), but the conditions do not impose any particular "default" value.

Remark 5.16. One may wonder that the natural number memory structure does not treat the number 0 as a value stored in a memory, since we define $i \notin \mathrm{supp}(\mathrm{m})$ if $\mathrm{m}(i) = 0$. It is not the case because we do not make use of a memory structure *alone*: in later sections, another structure (net, machine, or language) always come together with a memory structure, and an address with such a default value and an "empty" address are distinguished by whether the address is *linked* to some object (a 1 node, a position, or a variable) or not.

5.2.2 Instance: Probabilistic Memory

Since test function is defined to be probabilistic, it is straightforward to define a memory structure that probabilistically behaves. A common probabilistic operation is the (fair or biased) coin flip. A possible instance of memory structure that realizes the behavior of such an operation can be spelled out as:

- $\mathcal{I} = \mathbb{N}$,
- $\mathrm{Mem}_{\mathrm{P}} = \{\mathrm{m} \colon \mathcal{I} \to [0,1] \uplus \mathbb{N}\}$ with $\mathrm{supp}(\mathrm{m}) = \{i \in \mathcal{I} \mid \mathrm{m}(i) \neq 0\}$,
- $(\sigma \cdot \mathrm{m})(i) := \mathrm{m}(\sigma(i))$,
- $\mathcal{L} = \{\mathtt{succ}, \mathtt{pred}, \mathtt{coin}\}$, with $\mathrm{arity}(\mathtt{succ}) = \mathrm{arity}(\mathtt{pred}) = \mathrm{arity}(\mathtt{pred}) = 1$,
- update is defined as follows.
  - $\mathrm{update}(i, \mathtt{succ}, \mathrm{m}) = \mathrm{m}\{i{:=}\mathrm{m}(i) + 1\}$
  - $\mathrm{update}(i, \mathtt{pred}, \mathrm{m}) = \mathrm{m}\{i{:=}\max(0, \mathrm{m}(i) - 1)\}$
  - $\mathrm{update}(i, \mathtt{coin}, \mathrm{m}) = \mathrm{m}\{i{:=}1/2\}$
- $\mathrm{test}(i, \mathrm{m}) = \begin{cases} \{(\mathtt{true}, \mathrm{m})^{\mathrm{m}(i)}, (\mathtt{false}, \mathrm{m})^{1-\mathrm{m}(i)}\} & \text{if } \mathrm{m}(i) \leq 1 \\ \{(\mathtt{true}, \mathrm{m})^{1}\} & \text{if } \mathrm{m}(i) > 1. \end{cases}$

Intuitively, a value $\mathrm{m}(i) \in [0,1]$ on the address i in the memory m represents a probability to yield `true` when test is applied on the address.

Example 5.17. Now a memory $\mathrm{m}_0 = (0, 0, 0, \ldots)$ can be seen as a memory filled with the value "false". if by applying `coin` to the address 0, the memory goes into another memory $\mathrm{m}_1 = \mathrm{update}(0, \mathtt{coin}, \mathrm{m}_0) = (\frac{1}{2}, 0, 0, 0, \ldots)$. Then applying test, we obtain $\mathrm{test}(0, \mathrm{m}_1) = \{(\mathtt{false}, \mathrm{m}_1)^{\frac{1}{2}}, (\mathtt{true}, \mathrm{m}_1)^{\frac{1}{2}}\}$.

Remark 5.18. Precisely speaking, the memory structure defined above only accommodates binary branching while the language PPCF defined in [26] has `rand` function that yields $k < n$ with probability $1/n$ when applied to n. However such a function can be defined only using fair coin as shown in e.g. [21], using recursion that can be expressed in our framework. Moreover, in the same paper [26] they show that the choice among such probabilistic primitives is irrelevant with respect to observational equality, thus the difference is in any case not essential.

5.2.3 Instance: Quantum Memory

A notable instance of the notion of memory structure is a quantum memory. A standard model for quantum computation is the QRAM model: quantum data is stored in a memory seen as a list of (quantum) registers, each one holding a qubit which can be acted upon. The model supports three main operations: creation of a new register, measurement of a register, and application of unitary gates on one or more registers, depending on the arity of the gate under scrutiny. This model has been used extensively in the context of quantum lambda calculi [17, 79, 93], with minor variations. The main choice to be made is whether measurement is destructive (i.e., if one uses garbage collection) or not (i.e., the register is not reclaimed).

To fix things, we shall concentrate on the presentation given in [79]. We briefly recall it. Given n qubits, a memory is a normalized vector in $(\mathbb{C}^2)^{\otimes n}$ (equivalent to a ray). A linking function maps the position of each qubit in the list to some pointer name. The creation of a new qubit turns the memory $\phi \in (\mathbb{C}^2)^{\otimes n}$ into $\phi \otimes |0\rangle \in (\mathbb{C}^2)^{\otimes (n+1)}$. The measurement is destructive: if $\phi = \alpha_0 q_0 + \alpha_1 q_1$, where each q_b (with $b = 0, 1$) is normalized of the form $\sum_i \phi_{b,i} \otimes |b\rangle \otimes \psi_{b,i}$, then measuring ϕ returns $\sum_i \phi_{b,i} \otimes \psi_{b,i}$ with probability $|\alpha_b|^2$. Finally, the application of a k-ary unitary gate U on $\phi \in (\mathbb{C}^2)^{\otimes n}$ simply applies the unitary matrix corresponding to U on the vector ϕ. The language comes with a chosen set $\mathcal{U}$ of such gates.

The quantum memory can be presented using a memory structure: in the following we shall refer to the memory structure as $\mathrm{Mem}_{\mathcal{Q}}$. Let $\mathcal{F}_0$ be the set of (set-)maps from $\mathcal{I}$ to $\{0, 1\}$ that have value 0 everywhere except for a finite subset of $\mathcal{I}$. The quantum memory structure $\mathrm{Mem}_{\mathcal{Q}} = (\mathrm{Mem}_{\mathcal{Q}}, \cdot, \mathcal{I}, \mathcal{L})$ is defined as follows.

- $\mathcal{I} = \mathbb{N}$,
- $\mathrm{Mem}_{\mathcal{Q}} = \mathcal{H}_0$, that is the Hilbert space built from finite (complex) linear combinations over $\mathcal{F}_0$, with $\mathrm{supp}(\mathrm{m}) = \{i \in \mathcal{I} \mid \mathrm{m}(i) \neq 0\}$,
- $(\sigma \cdot \mathrm{m})(i) := \mathrm{m}(\sigma(i))$,
- $\mathcal{L}$ is a fixed set of unitary gates, with $\mathrm{arity}(\mathsf{U})$ being equal to the arity of the unitary gate $\mathsf{U} \in \mathcal{L}$,
- update and test are defined via the equivalence described below.

The operation update corresponds to application of an unitary gate to a quantum state, and test corresponds to measurement followed by a (classical) boolean test on the result.

Let $\mathrm{m} \in \mathrm{Mem}_{\mathcal{Q}}$. Since m has a finite support, a finite subset $\mathcal{I}_0 = \{i \mid \mathrm{m}(i) \neq 0\} \subseteq \mathcal{I}$ can always be taken. As fresh values are represented by 0 in m, m can be regarded as a superposition of some sequences that are equal to 0 on $I \setminus I_0$. Then m can be represented as "$\phi \otimes |000\ldots\rangle$" for some (finite) vector ϕ. By omitting the last $|0000\ldots\rangle$, now the vector ϕ is the standard presentation of finite-dimensional quantum state. The operations update and test can then be defined on the nominal set presentation through this equivalence.

The equations required for memory structures are indeed satisfied by $\mathrm{Mem}_{\mathcal{Q}}$. In this quantum setting, the meaning of the three groups of equations in Definition 5.14 are as follows. Equation 1. is simply renaming of qubits. 2. is a property known to hold when applying a unitary, and 3. holds because the equations correspond to the tensor of two unitaries or the tensor of a unitary and a measurement that is monoidal.

Remark 5.19. The reason why the set Mem is in the codomain of test: $\mathcal{I} \times \mathrm{Mem} \to \mathbb{B} \times \mathrm{Mem}$ is to accommodate the quantum instance. Measurement of a quantum state may collapse the state globally (see Section 3.5), and thus the globally modified memory has to be returned together with the resulting boolean in order to continue computation further.

5.3 Program Net

As observed in the previous three subsections, the notion of memory structure covers some important effects that "choose" branches probabilistically. In this section we parameterize SMEYLL nets with a memory structure. To avoid inessential difference on addresses, the extended nets are quotiented by permutations over addresses.

Definition 5.20 (Raw Program Nets)**.** Given a memory structure $\mathrm{Mem} = (\mathrm{Mem}, \cdot, \mathcal{I}, \mathcal{L})$, a *raw program net* on Mem is a triple $((R, \mathrm{op}_R), \mathrm{ind}_R, \mathrm{m})$ where

- (R, op_R) is a decorated SMEYLL net (see Definition 4.67),
- $\mathrm{ind}_R \colon \mathrm{Input}_R \rightharpoonup \mathcal{I}$ is an injective partial function that is however total on the occurrences of $\bot$ in the conclusions of R,
- $\mathrm{m} \in \mathrm{Mem}$.

We often omit the op_R function and write $(R, \mathrm{ind}_R, \mathrm{m})$ for a raw program net. We require that the arity of each sync node s matches the arity of $\mathrm{op}_R(s)$. A 1 node a is said to be *active* if $\mathrm{ind}_R(a)$ is defined.

Definition 5.21 (Program Nets)**.** A *program net* is defined to be an equivalence class of raw program nets over permutations of the indexes, where the action of a permutation $\sigma \in \mathrm{FinBij}(\mathcal{I})$ is defined by $\sigma(R, \mathrm{ind}_R, \mathrm{m}) = (R, \sigma \cdot \mathrm{ind}_R, \sigma \cdot \mathrm{m})$ and the equivalence relation $\sim$ is defined by: $(R', \mathrm{ind}_{R'}, \mathrm{m}') \sim (R, \mathrm{ind}_R, \mathrm{m})$ if $(R', \mathrm{ind}_{R'}, \mathrm{m}') = \sigma(R, \mathrm{ind}_R, \mathrm{m})$ for some $\sigma \in \mathrm{FinBij}(\mathcal{I})$. Thus formally a program net can be written as an equivalence class $\mathbf{R} = [(R, \mathrm{ind}_R, \mathrm{m})]$.

Notation 5.22. We denote the set of program nets by $\mathcal{N}$. The correspondence between a 1 node and an index i by the partial function $\mathrm{ind}_{\mathbf{R}}$ is graphically shown by a dotted line connecting the 1 node and the index i as in Figure 5.3: the left 1 node a_1 is linked to the address $i = \mathrm{ind}_{\mathbf{R}}(a_1)$ and thus active, while the right 1 node a_2 is not (i.e. $\mathrm{ind}_{\mathbf{R}}(a_2)$ is undefined). A program net $[(R, \mathrm{ind}_R, \mathrm{m})]$ will be depicted as in Figure 5.4, showing the memory m in a rectangle next to the SMEYLL net R and using the convention in Figure 5.3 if some 1 nodes are active.

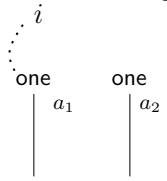


Figure 5.3: Partial Function $\mathrm{ind}_{\mathbf{R}}$, Graphically

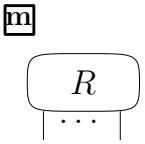


Figure 5.4: Program Net, Graphically

Example 5.23. An example of program net with the quantum memory defined in Section 5.2.3 is shown in Figure 5.5. The net corresponds to the translation[1] of a quantum program $\texttt{letrec}\, f\, x = (\texttt{if}\, x\, \texttt{then}\, \mathsf{new}\, \texttt{else}\, f\, (\mathsf{H}\, \mathsf{new}))\, \texttt{in}\, (f\, (\mathsf{H}\, \mathsf{new}))$, where the language and the translation will be made precise later (Section 5.5) in the chapter. Intuitively, the program flips a (fair) "quantum coin"; if it yields head, then a fresh 1-qubit quantum state is returned and the program halts. If it yields tail the program recursively flips a new coin until head comes out. Observe that each sync node is equipped with an operation (via op), and the net comes with a memory.

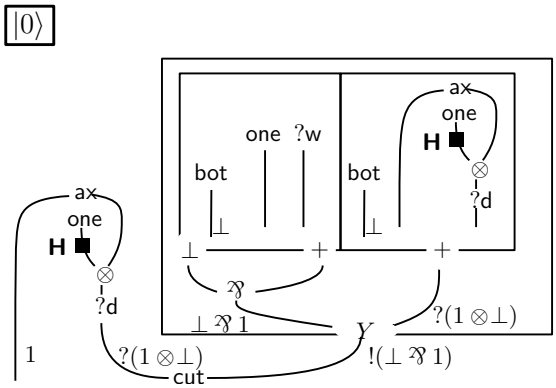


Figure 5.5: Example of Program Net

5.3.1 Reduction Rules

The reduction rules (as a PARS) for program nets are defined on raw program nets, and then the rules are shown to be compatible with the equivalence relation by permutations of indexes (Lemma 5.25). In Figure 5.7, we additionally have a rule to *link* an index of memory to a 1 node at surface, and the rules to remove sync nodes and to open multi ⊥-boxes now interact with the memory attached to the net. More precisely,

Definition 5.24 (Reduction Rules of Raw Program Nets)**.** The reduction rules of raw program nets are shown in Figure 5.6 and 5.7. Rules except $\rightsquigarrow_{\mathrm{test}}$ are meant to represent $(R, \mathrm{ind}_R, \mathrm{m}) \rightsquigarrow \{(R', \mathrm{ind}_{R'}, \mathrm{m}')^1\}$. The right-hand side of the $\rightsquigarrow_{\mathrm{test}(i)}$ rule denotes substitution: given $\mathrm{test}(i, \mathrm{m}) = \{(\texttt{true}, \mathrm{m}_0)^{p_0}, (\texttt{false}, \mathrm{m}_1)^{p_1}\}$, the right-hand side of the rule represents a distribution of raw program nets $\{(S_0, \mathrm{ind}_R, \mathrm{m}_0)^{p_0}, (S_1, \mathrm{ind}_R, \mathrm{m}_1)^{p_1}\}$.[2] The three rules in Figure 5.7 have to comply with the following conditions when applied:

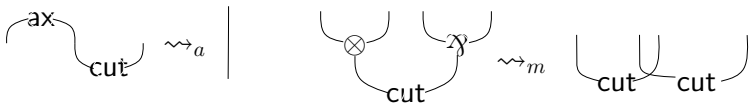
- The link(i) rule can be applied only if the index $i \in \mathcal{I}$ satisfies $i \notin \mathrm{supp}(\mathrm{m})$ and $i \notin \mathrm{Im}(\mathrm{ind}_R)$.
- The update(s) rule can be applied only if all the 1 nodes involved in the reduction are active (i.e. ind_R is defined on all of them).
- The test(i) rule can be applied only if the 1 node involved in the reduction is active.

[1] It is simplified from the exact translation in Section 5.5: all the ax-cut redexes are already reduced in the figure, and the terms in the "then" clause and "else" clause are not closed. This is not to complicate the example too much at the same time conveying some computational intuition by the example.

[2] Precisely speaking this is an abuse of notation, since we substitute a pair of type {SMEYLL nets} × ($\mathrm{Input}_R \rightharpoonup \mathcal{I}$) to a boolean of type $\mathbb{B}$. We will use a similar abuse of notation in the definition of transition rules of the multi-token machine (Definition 5.33).

We write $(R, \mathrm{ind}_R, \mathrm{m}) \overset{r}{\rightsquigarrow} \mu$ for the reduction of the redex r in the raw program net $(R, \mathrm{ind}_R, \mathrm{m})$.

Multiplicatives



Exponentials

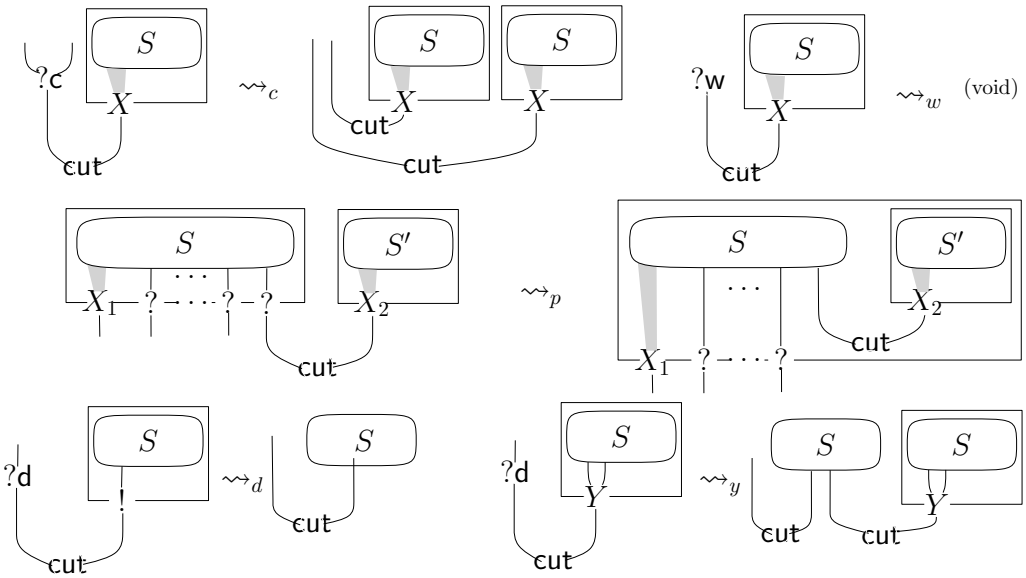


Figure 5.6: Rules Not Involving Memories

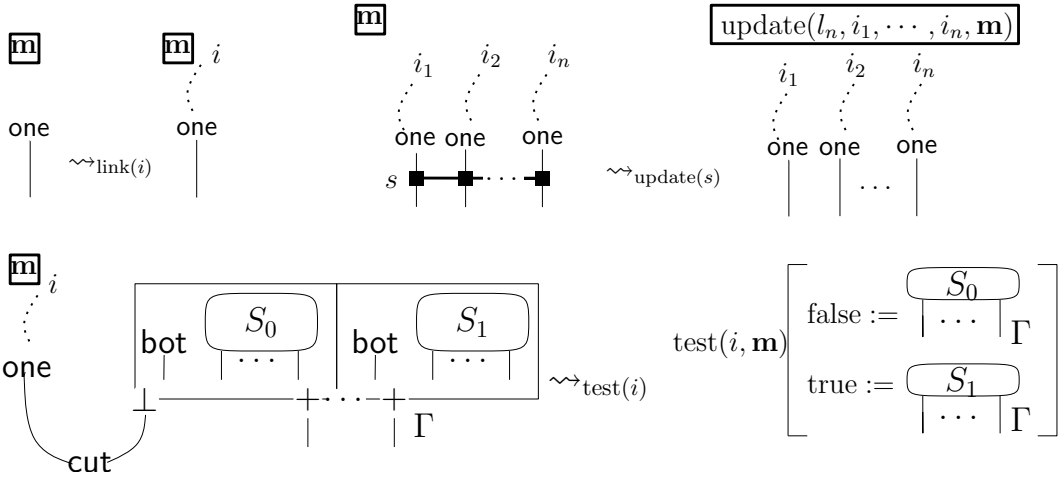


Figure 5.7: Rules Involving Memories

Lemma 5.25 (Preservation of Equivalence)**.** *Let* $(R, \mathrm{ind}_R, \mathrm{m}) \overset{r}{\rightsquigarrow} \mu$ *and* $(R, \sigma \cdot \mathrm{ind}_R, \sigma \cdot \mathrm{m}) \overset{r}{\rightsquigarrow} \nu$ *be two reductions of raw program nets on the same redex* r *in the underlying SMEYLL net* R*. Then* $\mu \sim \nu$ *pointwisely: for all* $\mathbf{S} \in \mathrm{supp}(\mu)$*, there exists a program net* $\mathbf{S}' \in \mathrm{supp}(\nu)$ *satisfying* $\mathbf{S} \sim \mathbf{S}'$ *and* $\mu(\mathbf{S}) = \nu(\mathbf{S}')$*; for all* $\mathbf{S}' \in \mathrm{supp}(\nu)$*, there exists a program net* $\mathbf{S} \in \mathrm{supp}(\mu)$ *satisfying* $\mathbf{S} \sim \mathbf{S}'$ *and* $\mu(\mathbf{S}) = \nu(\mathbf{S}')$*.*

Proof. Let us check the rule $\rightsquigarrow_{\mathrm{test}(i)}$. Suppose $(R', \mathrm{ind}_{R'}, \mathrm{m}') = \sigma(R, \mathrm{ind}_R, \mathrm{m})$, $(R, \mathrm{ind}_R, \mathrm{m}) \overset{r}{\rightsquigarrow}_{\mathrm{test}(i)} \mu = \{(R_0, \mathrm{ind}_{R_0}, \mathrm{m}_0)^{p_0}, (R_1, \mathrm{ind}_{R_1}, \mathrm{m}_1)^{p_1}\}$, and $(R', \mathrm{ind}'_R, \mathrm{m}') \overset{r}{\rightsquigarrow}_{\mathrm{test}(i)} \nu = \{(R'_0, \mathrm{ind}'_{R_0}, \mathrm{m}'_0)^{p'_0}, (R'_1, \mathrm{ind}_{R'_1}, \mathrm{m}'_1)^{p'_1}\}$ by reducing the same redex r. It suffices to show that $\nu = \sigma \cdot \mu$. Element-wisely, we have to check $R'_i = R_i$,

$\mathrm{ind}'_{R_i} = \sigma \circ \mathrm{ind}'_{R_i}$, $\mathrm{m}'_i = \sigma \cdot \mathrm{m}_i$, and $p'_i = p_i$ for $i \in \{0,1\}$. The first two follow by definition of $\rightsquigarrow_{\mathrm{test}(i)}$ and the last two follow from the equation $\sigma \cdot (\mathrm{test}(i)) = \mathrm{test}(\sigma(i), \sigma \cdot m)$. The other rules can be similarly checked. □

Since $(R, \mathrm{ind}_R, \mathrm{m}) \sim (R, \sigma \cdot \mathrm{ind}_R, \sigma \cdot \mathrm{m})$, we can regard the reductions as those on program nets. We write $\mathbf{R} \rightsquigarrow \mu$ for that reduction relation $\rightsquigarrow \subseteq \mathcal{N} \times \mathrm{Dist}(\mathcal{N})$ over program nets and distributions on program nets. In this way, the set of program nets and the reduction over program nets form a PARS. The PARS indeed satisfies the diamond property defined in section 5.1, and hence also satisfies the uniqueness of normal forms:

Lemma 5.26 (Diamond Property of Program Net)**.** *The probabilistic abstract reduction system $(\mathcal{N}, \rightsquigarrow)$ satisfies the diamond property.*

Lemma 5.26 follows from the following lemma.

Lemma 5.27 (Locality of $\rightsquigarrow$)**.** *Assume that $\mathbf{R} = [(R, \mathrm{ind}_R, \mathrm{m})]$ has two distinct redexes r_1 and r_2, with $\mathbf{R} \overset{r_1}{\rightsquigarrow} \mu_1$, $\mathbf{R} \overset{r_2}{\rightsquigarrow} \mu_2$ and $\mu_1 \neq \mu_2$. Then the redex r_2 (resp. r_1) is still a redex in each $(R', \mathrm{ind}_{R'}, \mathrm{m}') \in \mathrm{supp}(\mu_1)$ (resp. $\mathrm{supp}(\mu_2)$).*

Proof. By case analysis. □

Proof. (of Lemma 5.26.) Lemma 5.27 implies the following two facts:
(1) If $(R, \mathrm{ind}_R, \mathrm{m}) \rightsquigarrow \mu$ with $\mu^\circ \neq \emptyset$, then the raw program net $(R, \mathrm{ind}_R, \mathrm{m})$ contains exactly one redex.
(2) If $(R, \mathrm{ind}_R, \mathrm{m}) \overset{r_1}{\rightsquigarrow} \mu$ and $(R, \mathrm{ind}_R, \mathrm{m}) \overset{r_2}{\rightsquigarrow} \xi$ with $\mu \neq \xi$, then there exists ρ satisfying $\mu \rightrightarrows \rho$ and $\xi \rightrightarrows \rho$. Concretely, $\mu \rightrightarrows \rho$ is obtained by reducing the redex r_2 in each $(R', \mathrm{ind}_{R'}, \mathrm{m}') \in \mathrm{supp}(\mu)$, and $\xi \rightrightarrows \rho$ is obtained by reducing r_1.

Assuming $\mu \rightrightarrows \nu$ and $\mu \rightrightarrows \xi$, item 1. implies $\nu^\circ = \xi^\circ$, and item 2. implies $\exists \rho. \nu \rightrightarrows \rho \wedge \xi \rightrightarrows \rho$. Let us review some of the non-evident cases explicitly.

If r_1 and r_2 are both non-active $\mathsf{1}$ nodes, say x and y respectively, $(R, \mathrm{ind}_R, \mathrm{m})$ reduces to $(R, \mathrm{ind}_R \cup \{x \mapsto i, y \mapsto j\}, \mathrm{m})$ and $(R, \mathrm{ind}_R \cup \{x \mapsto k, y \mapsto l\}, \mathrm{m})$ for some fresh indexes i, j, k, l. The permutation $(i,k) \circ (j,l)$ renders the two program nets equivalent.

If both r_1 and r_2 modify memories (i.e. they perform either update or test), the property holds because the injectivity of ind_R guarantees that we always have the requirement (disjointness of indexes) of the equations. Hence the two reductions commute both on memory (up to group action) and on probability. □

Corollary 5.28 (Uniqueness of Normal Forms)**.** *The probabilistic abstract reduction system $(\mathcal{N}, \rightsquigarrow)$ satisfies the uniqueness of normal forms.*

Proof. By Lemma 5.26 and Theorem 5.12. □

Example 5.29. The program net in Example 5.23 reduces as shown in Figure 5.8 and Figure 5.9. The first reduction applies the Hadamard gate H to the corresponding memory via the index $0 = \mathrm{ind}(a)$ where a is the only $\mathsf{1}$ node at surface. Then after some more reduction, the multi $\perp$-box is opened by test rule that yields one of the two reducts probabilistically, here with the same probability $\frac{1}{2}$. One of the two reaches to a normal form after some more reduction with the only one $\mathsf{1}$ node is linked to a fresh index 1; the other continues to reduce by first assigning a fresh index 1 to the $\mathsf{1}$ node that newly came out by test reduction, then applying H to that $\mathsf{1}$ node (now the quantum memory is in the state $\frac{1}{\sqrt{2}}|1\rangle \otimes (|0\rangle + |1\rangle)$, as shown in Figure 5.9), and eventually the second test reduction will take place.

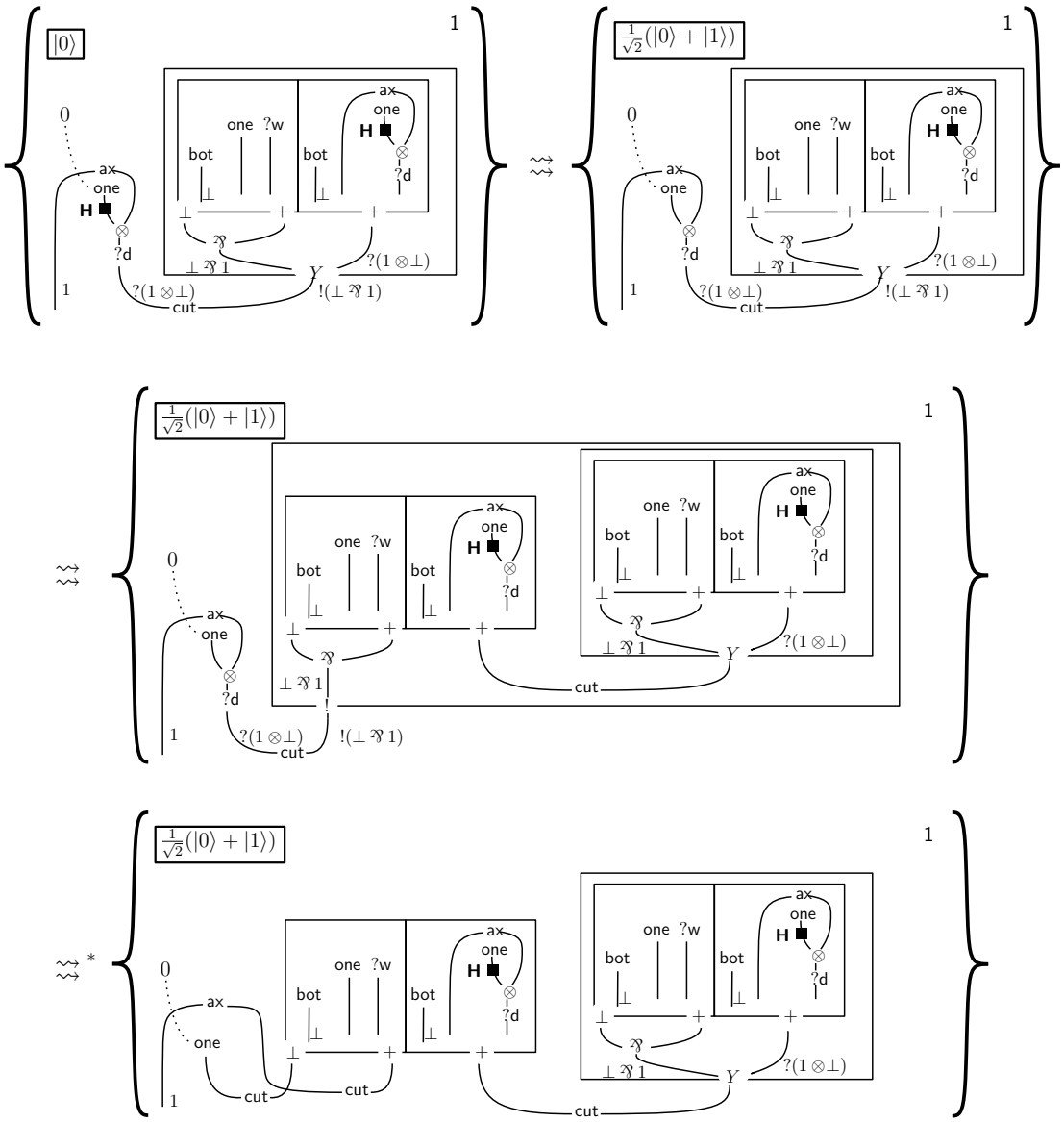


Figure 5.8: Example of Program Net Reduction (i)

5.4 Memory-Based Synchronous Interaction Abstract Machine

In the section we define the *Memory-based Synchronous Interaction Abstract Machine (MSIAM)* that is a multi-token machine whose states and the transitions are also equipped with memories and probabilities induced from the memory structure. The way we define them precisely follows the one we did in the previous section: first we define *raw states*, then *states* are defined as equivalence classes defined by permutations. Transition rules are first defined on raw states, then they naturally extend to those on the equivalence classes.

Definition 5.30 (Raw MSIAM States)**.** Given a memory structure $\mathrm{Mem} = (\mathrm{Mem}, \mathcal{I}, \mathcal{L})$ and a raw program net $(R, \mathrm{ind}_R, \mathrm{m}_R)$ on Mem, a *raw state* is a tuple $(\mathrm{T}, \mathrm{ind}_\mathrm{T}, \mathrm{m}_\mathrm{T})$ where

- T is a state of the SIAM $\mathcal{M}_R$,
- $\mathrm{ind}_\mathrm{T} \colon \mathrm{Start}_R \rightharpoonup \mathcal{I}$ is a partial injective function,
- $\mathrm{m}_\mathrm{T} \in \mathrm{Mem}$.

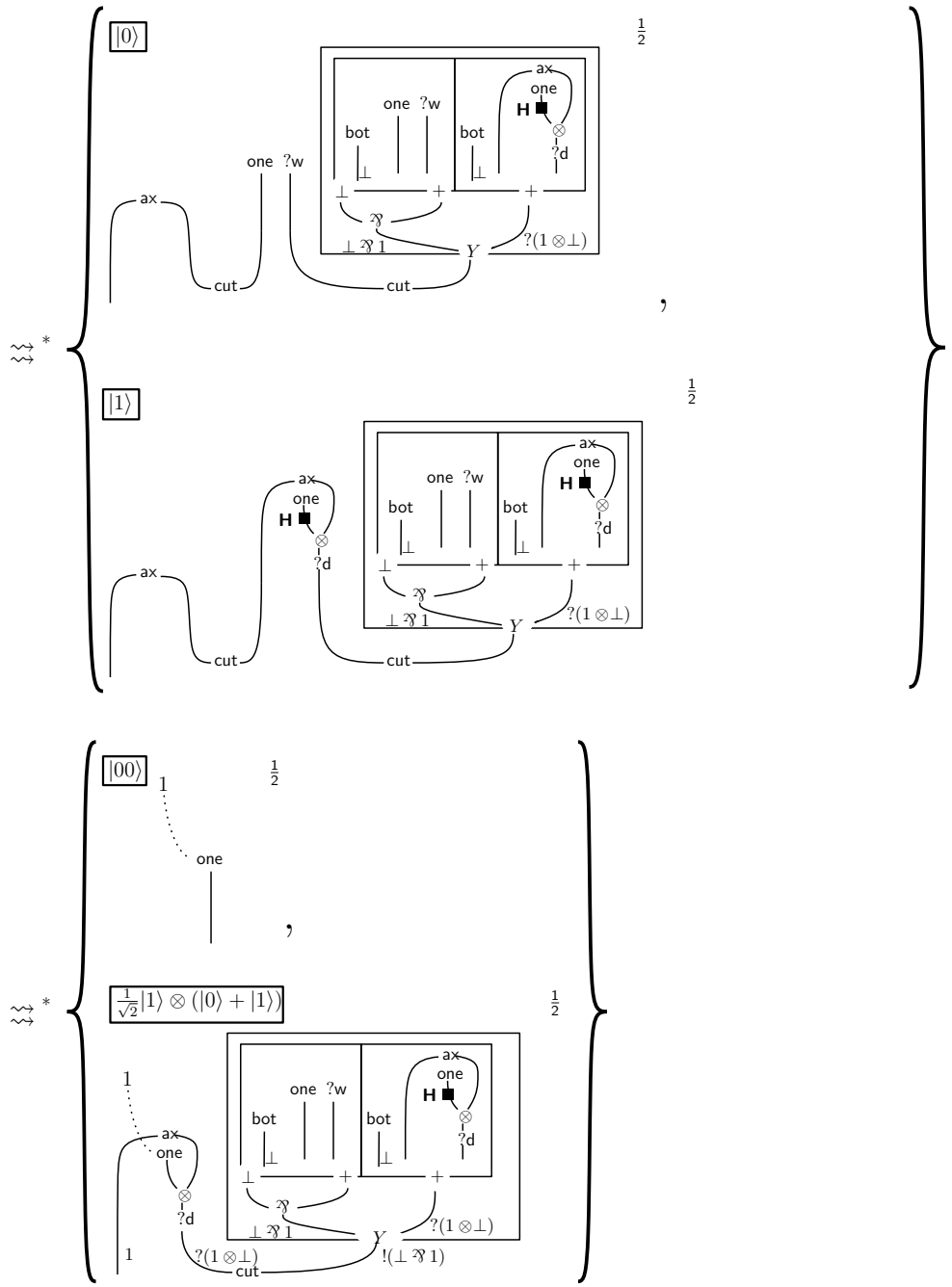


Figure 5.9: Example of Program Net Reduction (ii)

Definition 5.31 (MSIAM States)**.** An *MSIAM state* is defined as an equivalence class $\mathbf{T} = [(\mathrm{T}, \mathrm{ind}_\mathrm{T}, \mathrm{m}_\mathrm{T})]$ of raw states over permutations, with the action of FinBij($\mathcal{I}$) on tuples being the natural one, that is $\sigma \cdot (\mathrm{T}, \mathrm{ind}_\mathrm{T}, \mathrm{m}_\mathrm{T}) = (\mathrm{T}, \sigma \cdot \mathrm{ind}_\mathrm{T}, \sigma \cdot \mathrm{m}_\mathrm{T})$.

Notation 5.32. We depict an MSIAM state $[(\mathrm{T}, \mathrm{ind}_\mathrm{T}, \mathrm{m}_\mathrm{T})]$ on the program net $[(R, \mathrm{ind}_R, \mathrm{m}_R)]$ as in Figure 5.10. To distinguish from the memory of the underlying program net (which does not coincides with the memory of the MSIAM state in general), the memory m_T is surrounded by an ellipse. Notation for tokens is the same as the SIAM case.

Definition 5.33 (Transition Rules of MSIAM)**.** The transition rules on raw states are defined in Figure 5.11 and in Figure 5.12 with the same conventions

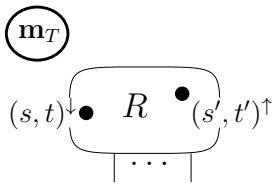


Figure 5.10: MSIAM State, Graphically

as the ones of the SIAM. The rules in Figure 5.11 do not involve memories and indexes: on the first components of raw states the rules acts in exactly the same way as those for the SIAM, and the second and the third components remain unchanged (thus the memories are omitted in Figure 5.11). Formally, if $\mathrm{T} \to \mathrm{U}$ in Figure 5.11, then $(\mathrm{T}, \mathrm{ind_T}, \mathrm{m}) \to \{(\mathrm{U}, \mathrm{ind_T}, \mathrm{m})^1\}$. The multi-token conditions to generate a token from a dereliction node and to move a token inside a multi $\perp$-box are again the same as the SIAM. The rules in Figure 5.12 interact with memories:

- The rule $\to_{\mathrm{link}(i)}$ generates a token associated to an address. Let $\mathbf{p}$ be the token in the right-hand side of the rule, a be the 1 node depicted in the figure, and c be the conclusion of a. Then, formally it defines a transition $(\mathrm{T}, \mathrm{ind_T}, \mathrm{m_T}) \to_{\mathrm{link}(i)} (\mathrm{U}, \mathrm{ind_T} \cup \{\mathrm{orig}_{\mathbf{p}} \mapsto i\}, \mathrm{m_T})$ where i is defined to be $\mathrm{ind}_{\mathbf{R}}(a)$ if a is active in the underlying program net $\mathbf{R}$, otherwise defined to be an address fresh in both $\mathrm{ind_T}$ and $\mathrm{m_T}$. Note that the memory m itself is unchanged by the rule.

- The rule $\to_{\mathrm{update}(l)}$ applies the function update when tokens simultaneously cross a sync node, with the same condition as the SIAM. Let $l \in \mathcal{L}$ be the operation associated to a sync node, n be its arity, and $\vec{i}$ be the sequence of addresses associated to the tokens, i.e. $\mathrm{ind_T}(\mathrm{orig_T}(\mathbf{p}_1))$, $\mathrm{ind_T}(\mathrm{orig_T}(\mathbf{p}_2)), \ldots, \mathrm{ind_T}(\mathrm{orig_T}(\mathbf{p}_n))$ where $\mathbf{p}_i$'s are the tokens depicted. Then $(\mathrm{T}, \mathrm{ind_T}, \mathrm{m_T}) \to_{\mathrm{update(l)}} \{(\mathrm{U}, \mathrm{ind_T}, \mathrm{update}(l, \vec{i}, \mathrm{m_T})^1\}$.

- The rule $\to_{\mathrm{test}(i)}$ is the only rule that (possibly) yields a distribution not in the form $\{(\mathrm{T}, \mathrm{ind_T}, \mathrm{m_T})^1\}$, using the function test. When a token $\mathbf{p} \in \mathrm{T}$ is at the conclusion of the principal conclusion of a multi $\perp$-box and $i = \mathrm{ind_T}(\mathrm{orig}(\mathbf{p}))$, then $(\mathrm{T}, \mathrm{ind_T}, \mathrm{m_T}) \to_{\mathrm{test}(i)} \mathrm{test}(i, \mathrm{m_T})[\texttt{false} := (\mathrm{T}_0, \mathrm{ind_T}), \texttt{true} := (\mathrm{T}_1, \mathrm{ind_T})]$ where T_0 (resp. T_1) is the SIAM state containing a token on the conclusion of the left (resp. right) bot node.

Example 5.34. The MSIAM of the program net in Example 5.23 starts from the initial state shown at top left in Figure 5.13. Then the token from the 1 node passes the sync node applying H gate to the corresponding address of the memory; the dereliction token meanwhile opens the Y-box. The token from the 1 node enters the Y-box and eventually reaches the principal conclusion of the multi $\perp$-box in the Y-box. It causes the test transition rule and yields a probability distribution of MSIAM states, shown at the bottom of Figure 5.13. The run of the machine continues as in Figure 5.14: the state with a stable token on the left content of the multi $\perp$-box generates a token from 1 node inside the $\perp$-box; this token goes out from the boxes, and finally reaches at the conclusion of the net. The other state with a stable token on the right content of the multi $\perp$-box generates a token from a 1 node, but this token will reach the Y node from right and starts the second recursion, eventually reaches at the principal conclusion of the multi $\perp$-box (shown in the last distribution in Figure 5.14) and will continue to another probabilistic branching.

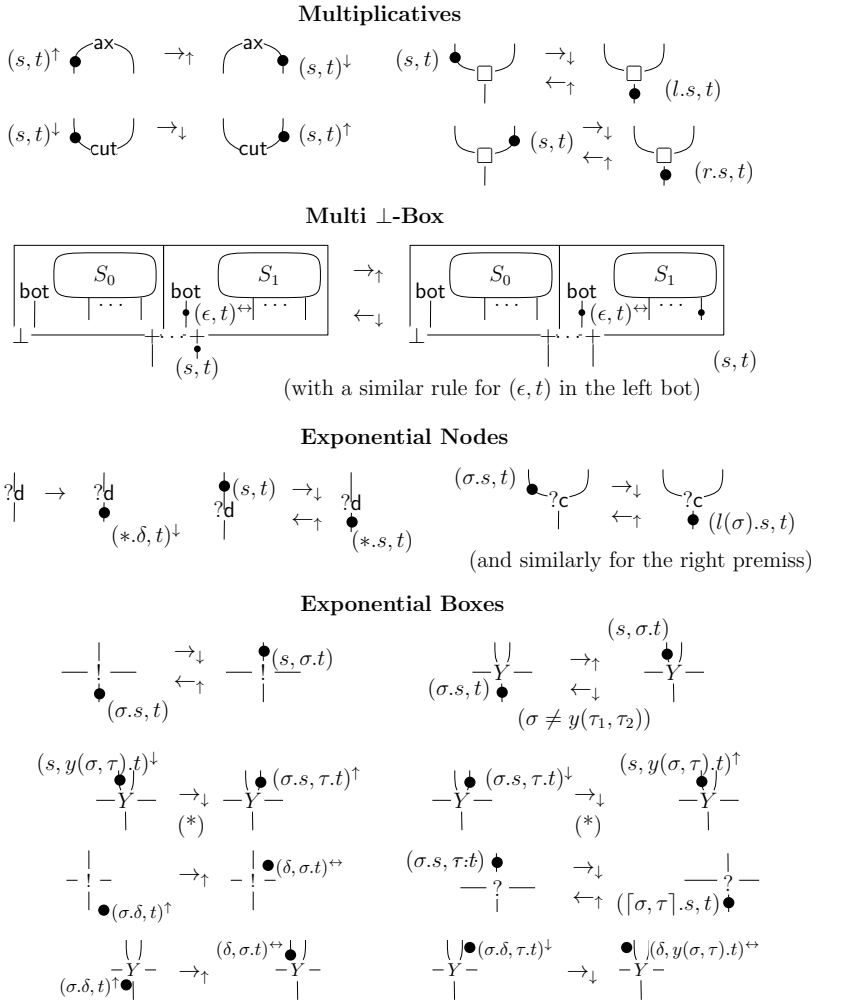


Figure 5.11: MSIAM Transitions Not Involving Memory

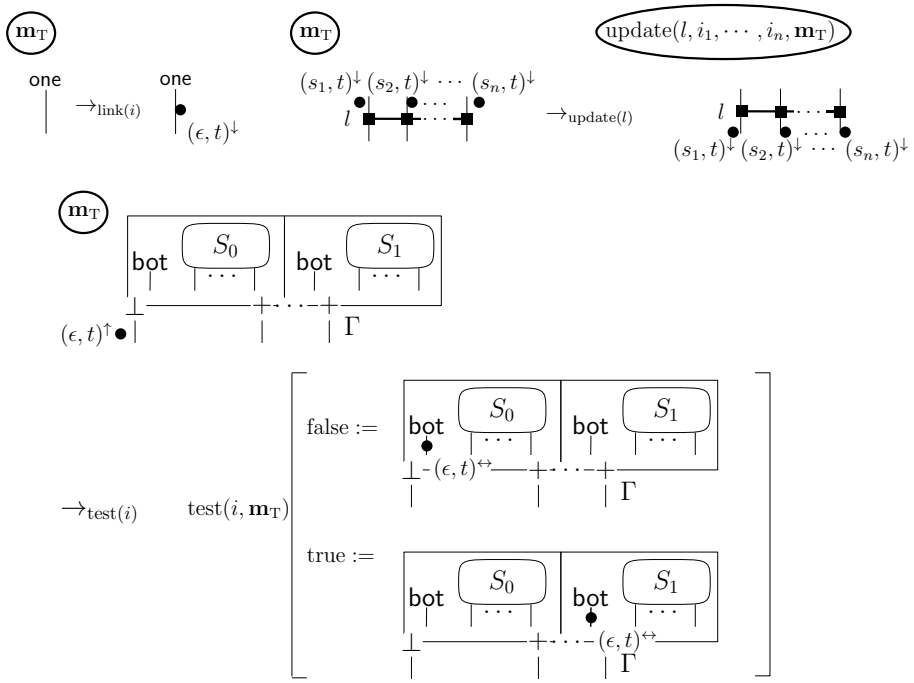


Figure 5.12: MSIAM Transitions Involving Memory

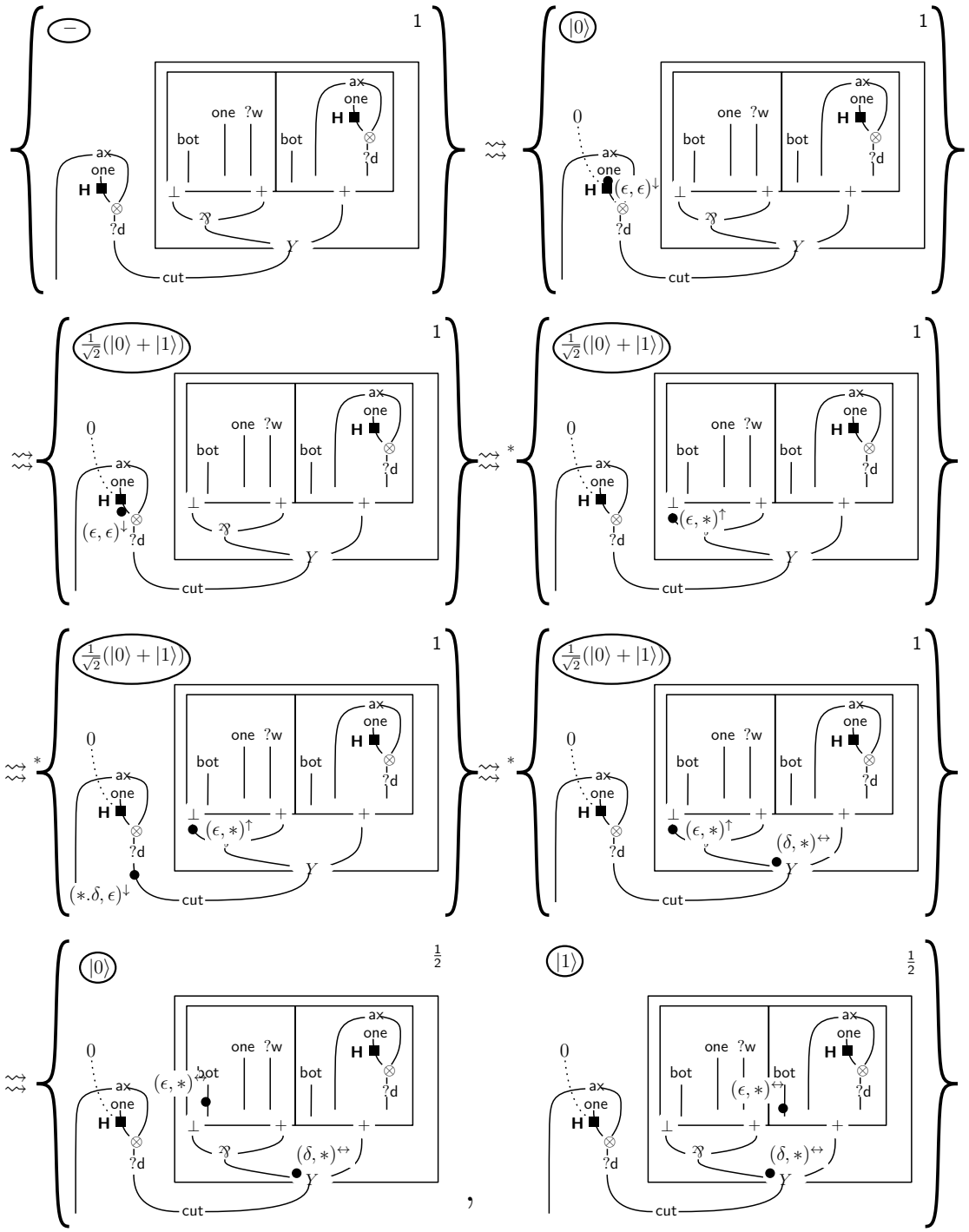


Figure 5.13: Example of MSIAM Transition (i)

Finally, we have three theorems, namely *invariance*, *deadlock-freedom*, and *adequacy*, where invariance and adequacy are in a quantitative form. Since the proofs are rather long, we defer them to the next two subsections.

Theorem 5.35 (Invariance of MSIAM)**.** *Let* $\mathbf{R}$ *be a program net of conclusion* 1 *and* $\mathbf{R} \rightsquigarrow \sum_i p_i \cdot \{\mathbf{R}_i\}$*. Then,* $\mathcal{M}_{\mathbf{R}}\Downarrow_q$ *if and only if* $\mathcal{M}_{\mathbf{R}_i}\Downarrow_{q_i}$ *for each* i *with* $\sum_i (p_i \cdot q_i) = q$*.*

Theorem 5.36 (Deadlock-Freedom of MSIAM)**.** *Let* $\mathbf{R}$ *be a program net of conclusion* 1*. If* $\mathbf{I}_{\mathbf{R}} \not\mapsto \mu$ *and* $\mathbf{T} \in \mathrm{supp}(\mu)$ *is terminal, then* $\mathbf{T}$ *is a* final *state.*

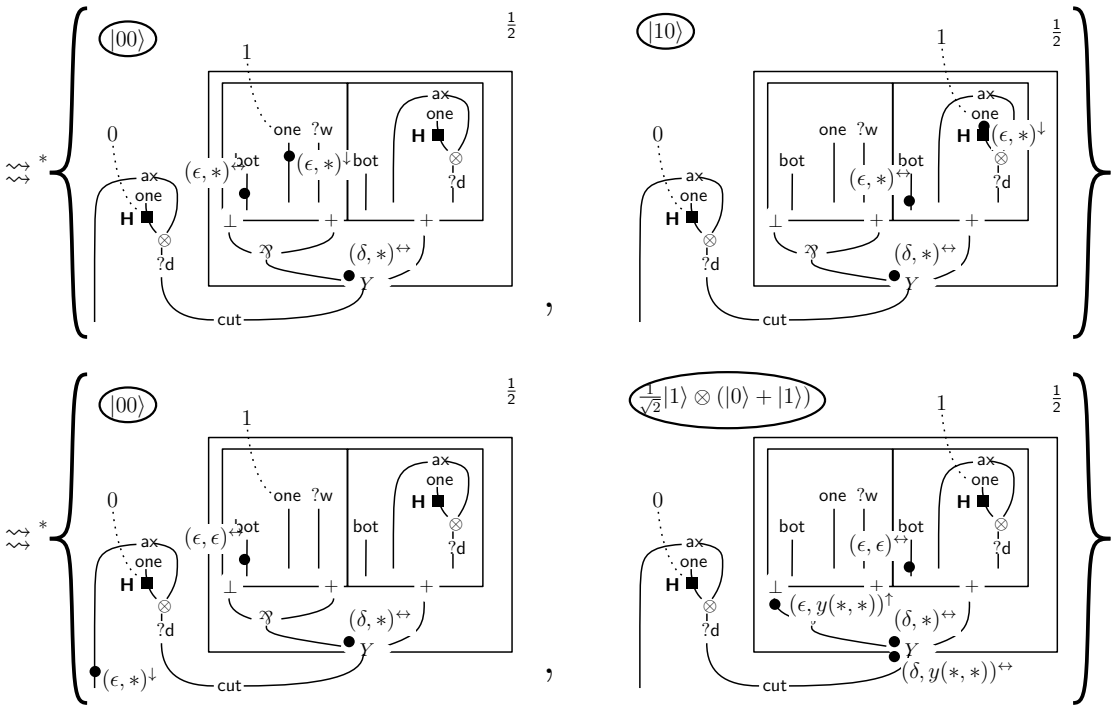


Figure 5.14: Example of MSIAM Transition (ii)

Theorem 5.37 (Adequacy of MSIAM)**.** *Let* $\mathbf{R}$ *be a program net of conclusion* 1*. Then,* $\mathcal{M}_{\mathbf{R}}\Downarrow_p$ *if and only if* $\mathbf{R}\Downarrow_p$*.*

5.4.1 Proofs of Invariance, Adequacy, and Deadlock-Freedom of MSIAM

The MSIAM is non-deterministic w.r.t. $\to$. However, by confluence and uniqueness of normal forms, any run from a distribution of states produces the same normal forms with the same probabilities. Therefore, we choose a run starting with specific transitions when we study termination of the MSIAM:

Remark 5.38 (Convention)**.** Let $\mathbf{R} \rightsquigarrow \rho$ be a program net reduction via link, update or test rule. Hereafter in this subsection, we will always look at a run in $\mathcal{M}_{\mathbf{R}}$ which begins as specified below:

1. link rule. Assume $(R, \mathrm{ind}_R, \mathrm{m}_R) \rightsquigarrow_{\mathrm{link(i)}} \{(R, \mathrm{ind}_R \cup \{x \mapsto i\}, \mathrm{m}_R)^1\}$. From the initial state $\mathbf{I_R}$ of the machine, first a transition using the link(i) rule occur on the 1 node involved in the reduction. We can choose the same address i because we know it is fresh in m_R. Therefore we have $(I, \mathrm{ind}_I, \mathrm{m}_R) \to_{\mathrm{link}(i)} \{(\mathrm{U}, \mathrm{ind}_U, \mathrm{m}_R)^1\} = \mu$.

2. update rule. Assume $(R, \mathrm{ind}_R, \mathrm{m}_R) \rightsquigarrow_{\mathrm{update}(s)} \{(R', \mathrm{ind}_R, \mathrm{update}(l, \vec{i}, \mathrm{m}_R)\}$. By definition of the reduction rule, the 1 nodes $\vec{a}$ in the redex are active; let $\vec{j}$ be the corresponding addresses $\mathrm{ind}(\vec{a})$. We choose a run starting with the transitions $(I, \mathrm{ind}_I, \mathrm{m}_R) \to_{\mathrm{link}(j_1)} \{(\mathrm{U}, \mathrm{ind}_U, \mathrm{m}_R)^1\}$ and $(\mathrm{U}, \mathrm{ind}_U, \mathrm{m}_R) \to_{\mathrm{update}(s)} \{(\mathrm{U}, \mathrm{ind}_U, \mathrm{update}(l, \vec{i}, \mathrm{m}_R)\} = \mu$.

3. test rule. Assume $(R, \mathrm{ind}_R, \mathrm{m}_R) \rightsquigarrow_{\mathrm{test}(j)} \rho$ where $R \rightsquigarrow_{u_i} R_i$ as SMEYLL nets, for each $i \in \{1, 2\}$. Again the 1 node a in the redex is active by definition. Let j be the corresponding address $\mathrm{ind}_{\mathbf{R}}(a)$. We start the run of the machine by applying a transition by link(j) rule to the initial state $(I, \mathrm{ind}_I, \mathrm{m}_R)$, then the token crosses the cut node in the redex,

and finally applies a transition by test(j) rule occur on the redex, reaching test$(j, \mathrm{m}_R)[\texttt{true} := \mathrm{U}_0, \texttt{false} := \mathrm{U}_1]) = \mu$.

We generalize the transformation map given in Section 4.5.3 to the one for the MSIAM. Let $\mathrm{trsf}_{R,r,R_i} : \mathrm{Pos}_R \rightharpoonup \mathrm{Pos}_{R_i}$ be the one defined on the SIAM states over SMEYLL nets with multi $\perp$-boxes.

We now extend $\mathrm{trsf}_{R \rightsquigarrow R_i}$ to MSIAM states as a generalization of the one given in Chapter 5. To do so smoothly, we define a subset $[\mathrm{trsf}_{R_i}]$ of the set of reachable states $\mathcal{S}_{\mathbf{I}_R}$ that depends on the reduction rule. Working with such states simplify the proofs, and is always possible by following the convention in Remark 5.38.

Definition 5.39. The set $[\mathrm{trsf}_{R_i}]$ is defined as follows.

- If the reduction is by $\rightsquigarrow_{\mathrm{link}(j)}$ rule, we define $[\mathrm{trsf}_{R_i}]$ as the set of states in $\mathcal{S}_{\mathbf{I}_R}$ in which $\mathrm{ind}(\mathbf{p}) = j$, where $\mathbf{p} \in \mathrm{SurfOne}_R$ is the position associated to the 1 node a.
- If the reduction is by $\rightsquigarrow_{\mathrm{update}(s)}$ rule, assume $\mathbf{p}_1, \ldots, \mathbf{p}_n$ are the positions associated to the premises of the sync node s (observe that each $\mathbf{p}_i$ belongs to Ones_R). $[\mathrm{trsf}_{R_i}]$ is defined to be the set of states $\mathbf{T} \in \mathcal{S}_{\mathbf{I}_R}$ satisfying $\{\mathbf{p}_1, \ldots, \mathbf{p}_n\} \subseteq \mathrm{orig}(T)$ and $\{\mathbf{p}_1, \ldots, \mathbf{p}_n\} \not\subseteq \mathrm{T}$.
- If the reduction is by $\rightsquigarrow_{\mathrm{test}(j)}$ rule, we define $[\mathrm{trsf}_{R_0}]$ as the set of states in $\mathcal{S}_{\mathbf{I}_R}$ that contain a token on the left $\perp$ node of the multi-$\perp$-box in the redex (the edge e_0 in Figure 4.32). We define $[\mathrm{trsf}_{R_1}]$ similarly.
- Otherwise we define $[\mathrm{trsf}_{R_i}] = \mathcal{S}_{\mathbf{I}_R}$.

Definition 5.40 (Transformation Map)**.** 1. $\mathrm{trsf}_i : [\mathrm{trsf}_{R_i}] \to \mathcal{S}_{R_i}$ maps the state $\mathbf{T} = [(T, \mathrm{ind}, \mathrm{m})]$ into $[\mathrm{trsf}_i(T, \mathrm{ind}, \mathrm{m})]$, with

$$\mathrm{trsf}_i(T, \mathrm{ind}, \mathrm{m}) = (\mathrm{trsf}_i(T), \mathrm{ind}, \mathrm{m}).$$

2. The definition extends linearly to distributions. Assume $\mu = \sum c_k \cdot \{\mathrm{T}_k\}$ and $\mathrm{T}_k \in [\mathrm{trsf}_{R_i}]$ for each T_k, then

$$\mathrm{trsf}_{R \rightsquigarrow R_i}(\mu) := \sum c_k \cdot \{\mathrm{trsf}_{R \rightsquigarrow R_i}(\mathrm{T}_k)\}.$$

We observe the following facts:

Fact 5.41. If $\mathbf{T} \in [\mathrm{trsf}_{R_i}]$, with $\mathbf{T} \to \mu$ and $\mathbf{S} \in \mathrm{supp}(\mu)$, then $\mathbf{S} \in [\mathrm{trsf}_{R_i}]$.

Fact 5.42. The construction given in 5.38 leads to a distribution μ in each case, where each state in the support $\mathrm{supp}(\mu)$ satisfies the following:

- $\mathbf{S}_i \in [\mathrm{trsf}_{R_i}]$.
- $\mathrm{trsf}_{R_i}(\mathbf{S}_i) = \mathbf{I}_{R_i}$.

Let us analyze the set of states that can be reached by a run of the MSIAM from the initial state, using the following notions:

Definition 5.43 ($\perp$-pair)**.** Given a multi $\perp$-box in a program net $\mathbf{R}$, let e_0 be the conclusion of the left $\perp$ node of the box and e_1 be the conclusion of the right $\perp$ node. For any stacks s, t, the pair of two stable positions (e_0, s, t) and (e_1, s, t) is called a $\perp$-*pair*.

The two positions in a $\perp$-pair are mutually exclusive in a state, because it must be the case that $\mathrm{orig}(e_0, s, t) = \mathrm{orig}(e_1, s, t)$.

Definition 5.44 (Conflict Relation)**.** Two states $\mathbf{T}, \mathbf{S} \in \mathcal{S}_{\mathbf{I}_R}$ are said to be *in conflict* and written $\mathbf{T} \smile \mathbf{S}$ if $\mathbf{T}$ contains one of the two positions of a $\perp$-pair and $\mathbf{S}$ contains the other.

We observe that conflict is hereditary with respect to transitions, because stable positions are never deleted or modified by a transition. Let $\mathbf{T}_{\rightrightarrows} = \{\mathbf{S} \mid \mathbf{T} \rightrightarrows^* \rho \wedge \mathbf{S} \in \mathrm{supp}(\rho)\}$. The following properties are all immediate to check:

Proposition 5.45. *Let* $\mathbf{R}$ *be a program net. In the PSIAM* $\mathcal{M}_{\mathbf{R}}$*,*

1. *If* $\mathbf{T} \smile \mathbf{T}'$*,* $\mathbf{S} \in \mathbf{T}_{\rightrightarrows}$*, and* $\mathbf{S}' \in \mathbf{T}'_{\rightrightarrows}$*, then* $\mathbf{S} \smile \mathbf{S}'$*.*
2. *If* $\mathbf{T} \to \mu$*, either* $\mu = \{\mathbf{T}'^1\}$*, or* $\mu = \{\mathbf{S}_0^{p_0}, \mathbf{S}_1^{p_1}\}$ *with* $\mathbf{S}_0 \smile \mathbf{S}_1$*.*
3. *If* $\mathbf{I}_{\mathbf{R}} \rightrightarrows^* \mu$*, then for each* $\mathbf{T} \neq \mathbf{T}' \in \mathrm{supp}(\mu)$*,* $\mathbf{T} \smile \mathbf{T}'$*.*

States in conflict are in particular disjoint: it is impossible for two states to reduces to the same state. Therefore we can sum them without any special care, unlike the example shown in Remark 5.5.

Lemma 5.46. *Let* $\mathbf{R}$ *be a program net and* $\mu \in \mathrm{Dist}(\mathcal{S}_{\mathbf{I}_R})$ *be a distribution of reachable states in* $\mathcal{M}_{\mathbf{R}}$*. The following rule is admissible in* $\mathcal{M}_{\mathbf{R}}$*:*

$$\frac{\forall \mathbf{T}_i, \mathbf{T}_j \in \mathrm{supp}(\mu).\ \mathbf{T}_i \smile \mathbf{T}_j \qquad \{\mathbf{T} \rightrightarrows^k \rho_{\mathbf{T}}\}_{\mathbf{T} \in \mathrm{supp}(\mu)}}{\mu \rightrightarrows^k \sum_{\mathbf{T} \in \mathrm{supp}(\mu)} \mu(\mathbf{T}) \cdot \rho_{\mathbf{T}}}\ .$$

As a consequence, the following also hold:

Corollary 5.47. *The following rule are admissible in* $\mathcal{M}_{\mathbf{R}}$*.*

$$\frac{\mathbf{S} \to \mu \quad \{\mathbf{T} \rightrightarrows^k \rho_{\mathbf{T}}\}_{\mathbf{T} \in \mathrm{supp}(\mu)}}{\mathbf{S} \rightrightarrows^{k+1} \sum_{\mathbf{T} \in \mathrm{supp}(\mu)} \mu(\mathbf{T}) \cdot \rho_{\mathbf{T}}}$$

$$\frac{\mathbf{S} \rightrightarrows^n \mu \quad \{\mathbf{T} \rightrightarrows^k \rho_{\mathbf{T}}\}_{\mathbf{T} \in \mathrm{supp}(\mu)}}{\mathbf{S} \rightrightarrows^{n+k} \sum_{\mathbf{T} \in \mathrm{supp}(\mu)} \mu(\mathbf{T}) \cdot \rho_{\mathbf{T}}}$$

The Reachability Relation $\looparrowright$

The reachability relation $\looparrowright$ (defined in Section 5.1) is a useful tool in the study of the MSIAM. In the case of the MSIAM, the relations $\looparrowright$ and $\rightrightarrows$ are equivalent with respect to normal forms. Note that this does not follow for a general PARS: we use facts about the conflict relation in the next proof.

Lemma 5.48. *Let* $\mathbf{R}$ *be a program net. If* $\{\mathbf{T}\} \rightrightarrows^* \xi$ *in* $\mathcal{M}_{\mathbf{R}}$ *then* $\mathbf{T} \looparrowright \xi$ *in* $\mathcal{M}_{\mathbf{R}}$*. Conversely, if* $\mathbf{T} \looparrowright \mu$ *in* $\mathcal{M}_{\mathbf{R}}$*, then there exists* $\rho \in \mathrm{Dist(A)}$ *satisfying* $\{\mathbf{T}\} \rightrightarrows^* \rho$ *and* $\mu^\circ \subseteq \rho^\circ$ *in* $\mathcal{M}_{\mathbf{R}}$*.* □

Proof. The former part is by induction on the length n of the sequence $\{\mathbf{T}\} \rightrightarrows^n \xi$. The latter is shown as follows. The leaves of a derivation tree of $\mathbf{T} \looparrowright \mu$ consists of finitely many transitions; let $\mathcal{A}$ be the set $\{\mathbf{T}_1 \to \mu_1, \mathbf{T}_2 \to \mu_2, \ldots, \mathbf{T}_k \to \mu_k\}$ of those transitions. The set $\mathcal{A}$ in particular contains finitely many transitions by test rules $\mathbf{S}_1 \to \{\mathbf{U}_{11}^{p_{11}}, \mathbf{U}_{12}^{p_{12}}\}, \mathbf{S}_2 \to \{\mathbf{U}_{21}^{p_{21}}, \mathbf{U}_{22}^{p_{22}}\}, \ldots, \mathbf{S}_l \to \{\mathbf{U}_{l1}^{p_{l1}}, \mathbf{U}_{l2}^{p_{l2}}\}$. By definition of $\looparrowright$, any state in $\mathrm{supp}(\mu)$ has a probability in the form of $q_1 q_2 \cdots q_h$

for some $h < k$ where $q_j \in \{p_{11}, p_{12}, \ldots p_{l1}, p_{l2}\}$ and satisfying that if $q_j = p_{i1}$, $q_{j'} = p_{i'1}$, and $j \neq j'$, then $i \neq i'$. Hence we can construct a sequence of derivations $\{\mathbf{T}^1\} = \mu_0 \rightrightarrows \mu_1 \rightrightarrows \mu_2 \cdots \rightrightarrows \mu_n$ in the following way: for each state in $\bar{\mu_i}$ that is also in the redex in $\mathcal{A}$, choose the corresponding transition in $\mathcal{A}$ in the derivation of $\mu_i \rightrightarrows \mu_{i+1}$; for any other state in $\bar{\mu_i}$, choose an arbitrary transition. In such a derivation, no undesirable "join" of states (see Remark 5.5) happens since all the states in any distribution μ_i are in conflict. Every terminal state reached in this way either matches one in μ including its probability or not in $\mathrm{supp}(\mu)$ by construction. Hence $\mu^\circ \subseteq {\mu_n}^\circ$. □

We also define another auxiliary relation $\mathbf{T} \not\mapsto^\circ \tau$ that is helpful in some proofs. This relation states that $\mathbf{T}$ reaches a set τ of terminal states. It is immediate that $\mathbf{T} \not\mapsto^\circ \tau$ if and only if $\exists \rho.\ \mathbf{T} \rightrightarrows^* \rho$ and $\tau \subseteq \rho^\circ$.

Definition 5.49 (The Relation $\not\mapsto^\circ$)**.** A relation $\not\mapsto^\circ \subseteq \mathcal{S}_{\mathbf{I}_R} \times \mathrm{Dist}(\mathcal{S}_{\mathbf{I}_R})$ is defined by $\mathbf{T} \not\mapsto^\circ \tau$ if there exists μ satisfying $\mathbf{T} \not\mapsto \mu$ and $\tau \subseteq \mu^\circ$.

Properties of trsf

We now study the action of trsf on transitions. We first look at how trsf maps initial/final/deadlock states. In short, the map trsf preserves those properties provided that the states belong to the set $[\mathrm{trsf}_{R_i}]$.

Lemma 5.50. *1. If* $\mathbf{I_R} \in [\mathrm{trsf}_{R_i}]$*, then* $\mathrm{trsf}_{R_i}(\mathbf{I_R}) = \mathbf{I}_{\mathbf{R}_i}$.

2. *Assume* $\mathbf{T} \in [\mathrm{trsf}_{R_i}]$ *is a final/deadlock state of* $\mathcal{M}_R$*; then* $\mathrm{trsf}_{R_i}(\mathbf{T})$ *is a final/deadlock state of* $\mathcal{M}_{R_i}$*.*

3. *If* $\tau = \tau^\circ$ *(i.e. all states in the distribution* τ *are terminal), and* $\mathrm{supp}(\tau) \subseteq [\mathrm{trsf}_{R_i}]$*, then* $\mathcal{T}(\tau) = \mathcal{T}(\mathrm{trsf}_{R_i}(\tau))$.

Another important property of the map trsf is that it preserves conflicts:

Lemma 5.51. *If* $\mathbf{T} \smile \mathbf{T}'$ *and* $\mathbf{T}, \mathbf{T}' \in [\mathrm{trsf}_{R_i}]$*, then* $\mathrm{trsf}_{R_i}(\mathbf{T}) \smile \mathrm{trsf}_{R_i}(\mathbf{T}')$

Fact 5.52 (Stable Tokens)**.** For any trsf_{R_i}, $S(\mathbf{T}) \geq S(\mathrm{trsf}_{R_i}(\mathbf{T}))$. Moreover, if the reduction $\rightsquigarrow$ is by d, y, or u_i rule, then $S(\mathbf{T}) > S(\mathrm{trsf}_{R_i}(\mathbf{T}))$.

We prove the following result from which invariance (Theorem 5.35) follows.

Lemma 5.53. *Assume* $\mathbf{R} \rightsquigarrow \sum_i p_i \cdot \{\mathbf{R}_i\}$. $\mathbf{I_R}$ *q-terminates if and only if* $\mathbf{I}_{\mathbf{R}_i}$ q_i*-terminates and* $\sum(p_i \cdot q_i) = q$.

Let us first sketch the proof. We need to work our way "back and forth" via Lemmas 5.56 and 5.59, because of the following observations.

Remark 5.54.

- Unfortunately, it is in general not true that $\mathbf{I_R} \rightrightarrows^* \mu$ implies $\mathrm{trsf}_{R_i}(\mathbf{I_R}) \rightrightarrows^* \mathrm{trsf}_{R_i}(\mu)$. However for the relation $\not\mapsto$ it is the case: if $\mathbf{I_R} \not\mapsto \mu$ in $\mathcal{M}_R$, then $\mathrm{trsf}_{R_i}(\mathbf{I_R}) \not\mapsto \mathrm{trsf}_{R_i}(\mu)$ holds (under natural conditions). This will be made precise by Lemma 5.56 later in this section.

- On the other side, the strength of the relation $\rightrightarrows$ is that if $\mathbf{I_R} \rightrightarrows^n \mu$, then for any sequence of the same length $\mathbf{I_R} \rightrightarrows^n \rho$, we have that $\rho^\circ = \mu^\circ$ thanks to uniqueness of normal forms (Theorem 5.12). This is not the case for the relation $\not\mapsto$ which is *not informative*. The (slightly complex) construction which is given by Lemma 5.59 allows us to exploit the power of $\rightrightarrows$.

As in the SIAM case, we say a transition $\mathbf{T} \to \{\mathbf{S}^1\}$ is a *collapsing transition* if $\mathrm{trsf}(\mathbf{T}) = \mathrm{trsf}(\mathbf{S})$, and the following lemma similarly holds.

Fact 5.55. Let $\mathbf{R} \overset{r}{\rightsquigarrow} \sum \mathbf{R}_i$ be a program net reduction. Given a transition $\mathbf{T} \to \mu$ in $\mathcal{M}_{\mathbf{R}}$, if $\mathbf{T} \in [\mathrm{trsf}_{\mathbf{R}_i}]$, then either the transition collapses, or $\mathrm{trsf}_{\mathbf{R}_i}(\mathbf{T}) \to \mathrm{trsf}_{\mathbf{R}_i}(\mu)$ is a transition of $\mathcal{M}_{\mathbf{R}_i}$.

Lemma 5.56. *If* $\mathbf{T} \in [\mathrm{trsf}_{R_i}]$ *and* $\mathbf{T} \not\mapsto \mu$ *(in* $\mathcal{M}_R$*), then* $\mathrm{trsf}_{R_i}(\mathbf{T}) \not\mapsto \mathrm{trsf}_{R_i}(\mu)$ *holds.*

Proof. We transform a derivation Π of $\mathbf{T} \not\mapsto \mu$ in $\mathcal{M}_R$ into a derivation of $\mathrm{trsf}_{R_i}(\mathbf{T}) \not\mapsto \mathrm{trsf}_{R_i}(\mu)$ in $\mathcal{M}_{R_i}$, by induction on the structure of the derivation.

- Case $\dfrac{}{\mathbf{T} \not\mapsto \{\mathbf{T}\}}$ becomes $\dfrac{}{\mathrm{trsf}_{R_i}\mathbf{T} \not\mapsto \{\mathrm{trsf}_{R_i}(\mathbf{T})\}}$
- Case $\dfrac{\mathbf{T} \to \sum p_{\mathbf{S}} \cdot \mathbf{S} \quad \{\overset{\cdots}{\overline{\mathbf{S} \not\mapsto \mu_{\mathbf{S}}}}\}}{\mathbf{T} \not\mapsto \sum p_{\mathbf{S}} \cdot \mu_{\mathbf{S}}}$

  We examine the left premise, checking if it collapses:

  - If it does not collapse, $\mathrm{trsf}_{R_i}(\mathbf{T}) \to \sum p_{\mathbf{S}} \cdot \mathrm{trsf}_{R_i}(\mathbf{S})$ is a transition of $\mathcal{M}_{R'}$ and we have:

    $$\frac{\mathrm{trsf}_{R_i}(\mathbf{T}) \to \sum p_{\mathbf{S}} \cdot \mathrm{trsf}_{R_i}(\mathbf{S}) \qquad \{\mathrm{trsf}_{R_i}(\mathbf{S}) \not\mapsto \mathrm{trsf}_{R_i}(\mu_{\mathbf{S}})\} \text{ by I.H.}}{\mathrm{trsf}_{R_i}(\mathbf{T}) \not\mapsto \sum p_{\mathbf{S}} \cdot \mathrm{trsf}_{R_i}(\mu_{\mathbf{S}})}$$

  - If it collapses, we must have $\mathbf{T} \to \{\mathbf{S}\}$, we also have $\mathrm{trsf}_{R_i}(\mathbf{T}) = \mathrm{trsf}_{R_i}(\mathbf{S})$,and the derivation Π is of the form:

    $$\frac{\mathbf{T} \to \{\mathbf{S}\} \quad \overset{\cdots}{\overline{\mathbf{S} \not\mapsto \mu}}}{\mathbf{T} \not\mapsto \mu}$$

    By induction, $\mathrm{trsf}_{R_i}(\mathbf{S}) \not\mapsto \mathrm{trsf}_{R_i}(\mu)$, and therefore we conclude $\mathrm{trsf}_{R_i}(\mathbf{T}) = \mathrm{trsf}_{R_i}(\mathbf{S}) \not\mapsto \mathrm{trsf}_{R_i}(\mu)$.

□

Lemma 5.56, the construction in Remark 5.38, and Lemma 5.50 allow us to transfer termination from $\mathbf{I}_{\mathbf{R}}$ to $\mathbf{I}_{\mathbf{R}_i}$, and to prove one direction of Lemma 5.53. The other direction is more delicate.

Assume that $\mathbf{I}_{\mathbf{R}_i}$ q_i-terminates. It means that for a certain n, whenever $\mathbf{I}_{\mathbf{R}_i} \rightrightarrows^n \sigma$ then $\mathcal{T}(\sigma) \geq q$ by definition. Lemma 5.59 below constructs such a sequence $\mathbf{I}_{\mathbf{R}_i} \rightrightarrows \sigma_1 \rightrightarrows \ldots \rightrightarrows \sigma_n = \sigma$ that satisfies $\sigma = \mathrm{trsf}_{R_i}(\mu)$ for some distribution μ of states in $\mathcal{M}_{\mathbf{R}}$. This allows us to transfer the properties of termination of $\mathbf{I}_{\mathbf{R}_i}$ back to $\mathbf{I}_{\mathbf{R}}$, ultimately leading to the other direction of Lemma 5.53.

Lemma 5.57. *From any* $\mathbf{T} \in \mathcal{S}_{\mathbf{I}_R}$*, there can be at most a finite number of consecutive collapsing transitions.*

Proof. By case analysis on reduction rules. □

Lemma 5.58. *Assume* $\mathrm{trsf}_{R_i}(\mathbf{T})$ *is terminal. Then there exists a state* $\mathbf{T}'$ *satisfying that* $\mathbf{T} \not\mapsto \{\mathbf{T}'^1\}$*, that* $\mathbf{T}'$ *is terminal, and that* $\mathrm{trsf}_{R_i}(\mathbf{T}') = \mathrm{trsf}_{R_i}(\mathbf{T})$.

Proof. By case analysis on reduction rules. □

Lemma 5.59. *Assume* $\mathbf{R} \rightsquigarrow \sum\{\mathbf{R}_i^{p_i}\}$ *and* $\mathbf{T} \in [\mathrm{trsf}_{R_i}]$. *For any* $n \in \mathbb{N}$ *the following holds.*

1. *there exists a distribution* $\mu \in \mathrm{Dist}(\mathcal{S}_R)$ *such that* $\mathbf{T} \twoheadrightarrow \mu$ *and* $\mathrm{trsf}_{R_i}(\mathbf{T}) \rightrightarrows^n \mathrm{trsf}_{R_i}(\mu)$*;*
2. *moreover, we can choose* μ *so that it satisfies* $\mathcal{T}(\mu) = \mathcal{T}(\mathrm{trsf}_{R_i}(\mu))$.

Proof. 1. We construct μ and its derivation, by induction on n.

$n = 1$. There are three cases. Note that there cannot be a transition $\mathbf{T} \to \{\mathbf{S}_1^{p_1}, \mathbf{S}_2^{p_2}\}$ because $\mathbf{T} \in [\mathrm{trsf}_{R_i}]$.

- If $\mathbf{T}$ is terminal, then $\mathrm{trsf}_{R_i}(\mathbf{T})$ is terminal by Lemma 5.50, and $\mathrm{trsf}_{R_i}(\mathbf{T}) \rightrightarrows \mathrm{trsf}_{R_i}(\mathbf{T})$.
- If there exists μ satisfying $\mathbf{T} \to \mu$ that is non-collapsing. We have $\mathrm{trsf}_{R_i}(\mathbf{T}) \to \mathrm{trsf}_{R_i}(\mu)$ and thus $\mathrm{trsf}_{R_i}(\mathbf{T}) \rightrightarrows \mathrm{trsf}_{R_i}(\mu)$ by definition of $\rightrightarrows$.
- If all transitions from $\mathbf{T}$ are collapsing, for such a reduction we have that $\mathbf{T} \to \mathbf{T}'$ and $\mathrm{trsf}_{R_i}(\mathbf{T}) = \mathrm{trsf}_{R_i}(\mathbf{T}')$. We repeat the argument on $\mathbf{T}'$ until we find a state $\mathbf{S}$ that either is terminal or has a non-collapsing transition $\mathbf{S} \to \mu$. By Lemma 5.57, this repetition always terminates. The former case is reduced to the case of $\mathbf{T}$ being terminal. The latter gives $\mathbf{S} \twoheadrightarrow \mu$ and therefore $\mathbf{T} \twoheadrightarrow \mu$, and $\mathrm{trsf}_{R_i}(\mathbf{T}) = \mathrm{trsf}_{R_i}(\mathbf{S}) \twoheadrightarrow \mathrm{trsf}_{R_i}(\mu)$, hence $\mathrm{trsf}_{R_i}(\mathbf{T}) \rightrightarrows \mathrm{trsf}_{R_i}(\mu)$.

$n > 1$. By the induction hypothesis we assume that we have obtained a derivation of $\mathbf{T} \twoheadrightarrow \rho$ with $\mathrm{trsf}_{R_i}(\mathbf{T}) \rightrightarrows^{n-1} \mathrm{trsf}_{R_i}(\rho)$. We have that $\mathrm{trsf}_{R_i}(\rho) = \sum \rho(\mathbf{S}) \cdot \mathrm{trsf}_{R_i}(\mathbf{S})$ by definition. For each $\mathbf{S} \in \mathrm{supp}(\rho)$, we apply the base step and obtain a derivation of $\mathbf{S} \twoheadrightarrow \mu_{\mathbf{S}}$ with $\mathrm{trsf}_{R_i}(\mathbf{S}) \rightrightarrows \mathrm{trsf}_{R_i}(\mu_{\mathbf{S}})$. Putting things together, we obtain a derivation of $\mathbf{T} \twoheadrightarrow \sum \rho(\mathbf{S}) \cdot \mu_{\mathbf{S}}$ and $\mathrm{trsf}_{R_i}(\mathbf{T}) \rightrightarrows^n \sum \rho(\mathbf{S}) \cdot \mathrm{trsf}_{R_i}(\mu_{\mathbf{S}})$ follows by Lemma 5.46.

2. Let $\mathbf{T} \twoheadrightarrow \mu$ be the result obtained in the proof of item 1. above. Let $\{\mathbf{S}_k\}$ be the set of states in $\mathrm{supp}(\mu)$ such that $\mathrm{trsf}_{R_i}(\mathbf{S}_k)$ is terminal. This induces a partition of μ, namely $\mu = \rho + \sum c_k \cdot \{\mathbf{S}_k\}$ where ρ does not contain terminal states as its support. By Lemma 5.58, we can take a terminal state $\mathbf{S}'_k$ satisfying $\mathbf{S}_k \twoheadrightarrow \{\mathbf{S}'_k\}$ and $\mathrm{trsf}_{R_i}(\mathbf{S}'_k) = \mathrm{trsf}_{R_i}(\mathbf{S}_k)$ for each $\mathbf{S}_k$. Observe also that ρ does not contain any terminal state. Let $\nu = \sum c_k \cdot \{\mathbf{S}'_k\}$. We have by transitivity $\mathbf{T} \twoheadrightarrow (\rho + \nu)$, and $\mathrm{trsf}_{R_i}(\mathbf{T}) \rightrightarrows^n \mathrm{trsf}_{R_i}(\rho + \nu)$ because $\mathrm{trsf}_{R_i}(\rho + \nu) = \mathrm{trsf}_{R_i}(\mu)$. Moreover, we have $\mathcal{T}(\mathrm{trsf}_{R_i}(\rho + \nu)) = \mathcal{T}(\mathrm{trsf}_{R_i}(\nu)) = \sum c_k$ because $\mathrm{trsf}_{R_i}(\nu) = \sum c_k \cdot \mathrm{trsf}_{R_i}(\mathbf{S}'_k)$. We conclude by observing that $\mathcal{T}(\rho + \nu) = \mathcal{T}(\nu) = \sum c_k$.

□

Summing up, we now have all the elements to prove Lemma 5.53.

Proof. (of Lemma 5.53.) $\Rightarrow$. Follows from Lemma 5.56, by using the construction in Remark 5.38, Lemma 5.50, and linearity of trsf. Assume $\mathbf{I}_R \rightrightarrows^* \mu$, with μ° not empty, and that the machine starts as described in Remark 5.38 if $\rightsquigarrow$ is by link, update, or test rule. We observe that every state $\mathbf{T} \in \mathrm{supp}(\mu)$ is contained in $[\mathrm{trsf}_i]$ for some i. We can then prove that for each i there exists $\mu_i \in \mathrm{Dist}(\mathcal{S}_{\mathbf{I}_R})$ such that $\mathbf{I}_{R_i} \twoheadrightarrow \mathrm{trsf}_{R_i}(\mu_i)$, and such that $\nu = \sum_i p_i \cdot \mu_i$.

$\Leftarrow$. Follows from Lemma 5.59. We examine the only non-straightforward case. Assume $\mathbf{R} \rightsquigarrow_{\mathrm{test(i)}} \{\mathbf{R}_0^{p_0}, \mathbf{R}_1^{p_1}\}$. We choose a run of the machine which starts as

described in Remark 5.38; we have that $\mathbf{I_R} \twoheadrightarrow \sum p_i \cdot \{\mathbf{T}_i\}$, with $\mathrm{trsf}_{R_i}(\mathbf{T}_i) = \mathbf{I}_{\mathbf{R}_i}$ by Fact 5.42. By hypothesis, $\mathbf{I}_{\mathbf{R}_i}$ terminates with probability at least q_i; assume it does so in n steps. By using Lemma 5.59, we build a derivation $\mathbf{T}_i \twoheadrightarrow \mu_i$ such that $\mathrm{trsf}_{R_i}(\mathbf{T}_i) \rightrightarrows^n \mathrm{trsf}_{R_i}(\mu_i)$ and $\mathcal{T}(\mu_i) = \mathcal{T}(\mathrm{trsf}_{R_i}(\mu_i))$. By Theorem 5.12, $\mathcal{T}(\mathrm{trsf}_{R_i}(\mu_i)) \geq q_i$. Putting all together, we have that $\mathbf{I_R} \twoheadrightarrow \sum p_i \cdot \mu_i$, and $\mathbf{I_R}$ terminates with probability at least $\sum p_i \cdot q_i$. □

Now we can prove the soundness theorem.

Proof. (of Theorem 5.35.) Suppose $\sum_i (p_i \cdot q_i) > q$. Then there exist distributions μ_i that satisfy $\mathbf{I}_{\mathbf{R}_i} \rightrightarrows^* \mu_i$, $\mathcal{T}(\mu_i) = q_i'$, and $\sum_i (p_i \cdot q_i') > q$. By Lemma 5.53, $\mathbf{I_R}$ $(\sum_i (p_i \cdot q_i))$-terminates, which contradicts to $\mathcal{M}_{\mathbf{R}} \Downarrow_q$.
Suppose $\sum_i (p_i \cdot q_i) < q$. Since $\mathcal{M}_{\mathbf{R}} \Downarrow_q$ holds, $\mathbf{I_R}$ q'-terminates for some probability q' that satisfies $\sum_i (p_i \cdot q_i) < q' \leq q$. By Lemma 5.53, $\mathbf{I}_{\mathbf{R}_i}$ q_i'-terminates for some q_i' with $\sum_i (p_i \cdot q_i') = q' > \sum_i (p_i \cdot q_i)$. This implies either or both of q_i' are larger than q_i, which contradicts to $\mathcal{M}_{\mathbf{R}_i} \Downarrow_{q_i}$.
Hence $\sum_i (p_i \cdot q_i)$ necessarily coincides with q. □

5.4.2 Proof of Adequacy of MSIAM

We are now able to establish adequacy (Theorem 5.37) and deadlock-freedom (Theorem 5.36). Both are direct consequence of Proposition 5.61 below, which in turn follows form Lemma 5.53 and Fact 5.60, by finely exploiting the interplay between nets and the machine as in the SIAM case.

Fact 5.60. Let R be a SMEYLL proof net with conclusion 1 in normal form. By Corollary 4.31, R has no cuts, and therefore consists of simply a single 1 node. On such a simple net, $\mathcal{M}_R$ can only terminate in a final state: no deadlock is possible.

Proposition 5.61 (Mutual Termination, Probabilistically)**.** *Let* $\mathbf{R}$ *be a net with conclusion* 1*. The following are equivalent:*

1. $\mathbf{I_R}$ *q-terminates;*
2. $\mathbf{R}$ *q-terminates.*

Moreover, if $\mathbf{I_R} \twoheadrightarrow \mu$ *and* $\mathbf{T} \in \mathrm{supp}(\mu)$ *is terminal, then* $\mathbf{T}$ *is a* final *state.*

Proof. **(1. ⇒ 2.) and 3.** We prove that

$$\text{if } \mathbf{I_R} \twoheadrightarrow^\circ \tau, \text{ then } \begin{array}{l}(*)\ \mathbf{R} \text{ terminates with probability at least } \mathcal{T}(\tau),\\ (**)\ \text{all states in } \mathrm{supp}(\tau) \text{ are final.}\end{array}$$

The proof is by nested induction on the lexicographically ordered pair $(S(\nu), W(\mathbf{R}))$, where $W(\mathbf{R})$ is the weight of the cuts *at surface* of $\mathbf{R}$, and $S(\nu) = \sum_{\mathbf{T} \in \mathrm{supp}(\tau)} S(\mathbf{T})$ with $S(\mathbf{T})$ the number of stable tokens in $\mathbf{T}$ (Fact 5.52). Both parameters are finite.

We will largely use the following fact (immediate consequence of the definition of $\twoheadrightarrow^\circ$ and of Lemma 5.50): if $\mathbf{T} \twoheadrightarrow^\circ \nu$ in $\mathcal{M}_{\mathbf{R}}$ and $\mathbf{T} \in [\mathrm{trsf}_i]$, then $\mathrm{trsf}_i(\mathbf{T}) \twoheadrightarrow^\circ \mathrm{trsf}_i(\nu)$.

- If $\mathbf{R}$ has no reduction step, then $S(\nu) = W(\nu) = 0$ and $\mathcal{T}(\mathbf{R}) = 1$, by Fact 5.60, which trivially proves (*); (**) also follows from Fact 5.60.

- Assume $\mathbf{R} \rightsquigarrow \{\mathbf{R}'^1\}$ is not by test rule. (observe that this is a deterministic reduction). We have that $\mathbf{I}_{\mathbf{R}'} \not\mapsto^\circ \mathrm{trsf}(\nu)$, and $\mathcal{T}(\mathrm{trsf}(\nu)) \geq \mathcal{T}(\nu)$. By Fact 5.52, $S(\mathrm{trsf}(\nu)) \leq S(\nu)$. If $\mathbf{R} \rightsquigarrow_d \mathbf{R}'$, then $S(\mathrm{trsf}(\nu)) < S(\nu)$ since ν only contains terminal states. Otherwise $S(\mathrm{trsf}(\nu)) = S(\nu)$ but $W(\mathbf{R}') < W(\mathbf{R})$ because the step reduces a `cut` node at surface, and *does not open any box*. Hence by induction, $\mathbf{R}'$ terminates with probability at least $\mathcal{T}(\mathrm{trsf}(\nu)) \geq \mathcal{T}(\nu)$ (and therefore so does $\mathbf{R}$) and all states in $\mathrm{trsf}(\nu)$ are final, from which (**) holds by Lemma 5.50.2.

- Assume $\mathbf{R} \rightsquigarrow_{\mathrm{test}(i,\mathrm{m})} \sum p_i \cdot \{\mathbf{R}_i\}$. From $\mathbf{I}_{\mathbf{R}} \not\mapsto^\circ \nu$, by Lemma 5.48 we have that there is a distribution ρ satisfying $\mathbf{I}_{\mathbf{R}} \rightrightarrows^* \rho$ and $\nu \subseteq \rho^\circ$. Using the construction in Remark 5.38, we have $\mathbf{I}_{\mathbf{R}} \rightrightarrows^* \sum p_i \cdot \{\mathbf{T}_i\}$, which induces a partition of ν in $\nu = p_0 \cdot \nu_0 + p_1 \cdot \nu_1$ with $\mathbf{T}_i \not\mapsto^\circ \nu_i$ for each i. We have that $S(\nu_i) < S(\nu)$, and that $\mathbf{I}_{\mathbf{R}_i} \not\mapsto^\circ \mathrm{trsf}_i(\nu_i)$, because $\mathrm{trsf}_i(\mathbf{T}_i)$ is defined and therefore $\mathrm{trsf}_i(\mathrm{U})$ is defined for each state $\mathrm{U} \in \nu_i$. By Fact 5.52, $S(\mathrm{trsf}_i(\nu_i)) \leq S(\nu_i) < S(\nu)$, thus by induction $\mathbf{R}_i$ terminates with probability at least $\mathcal{T}(\mathrm{trsf}_i(\nu_i))$, and all states in $\mathrm{supp}(\mathrm{trsf}_i(\nu_i))$ are final. Therefore, $\mathbf{R}$ terminates with probability at least $\sum p_i \cdot \mathcal{T}(\mathrm{trsf}_i(\nu_i)) \geq \sum p_i \cdot \mathcal{T}(\nu_i) = \mathcal{T}(\nu)$ by Lemma 5.50.3, and all states in $\mathrm{supp}(\nu)$ are final by Lemma 5.50.2.

2. $\Rightarrow$ 1. By hypothesis, $\mathbf{R} \rightrightarrows^n \rho$ with $\mathcal{T}(\rho) \geq q$. We prove the implication by induction on n.

Case $n = 0$. The implication is true by Fact 5.60.

Case $n > 0$. Assume $\mathbf{R} \rightsquigarrow \sum p_i \cdot \mathbf{R}_i$. By hypothesis, each $\mathbf{R}_i$ terminates with probability at least q_i (with $\sum p_i \cdot q_i = q$). By induction, each $\mathbf{I}_{\mathbf{R}_i}$ q_i-terminates, and therefore (Lemma 5.53) $\mathbf{I}_{\mathbf{R}}$ q-terminates. □

Now the deadlock-freedom and adequacy can be shown by using the proposition above.

Proof. (of Theorem 5.36) Immediately follows from Proposition 5.61. □

Proof. (of Theorem 5.37) Assume $\mathcal{M}_{\mathbf{R}} \Downarrow_p$. Then $\mathbf{R}$ cannot q-terminate with $p < q$, since it implies that $\mathcal{M}_{\mathbf{R}}$ q-terminates by Proposition 5.61, which contradicts to $\mathcal{M}_{\mathbf{R}} \Downarrow_p$. Similarly $\mathbf{R}$ q'-terminates for any $q' < p$ by Proposition 5.61 since $\mathcal{M}_{\mathbf{R}}$ does so. Hence $\mathbf{R} \Downarrow_p$. The other direction is shown in exactly the same way. □

5.5 Memory-Based PCF

The language is more or less the same as in the one in chapter 4, but this time parameterized by memory structures. By interpreting the language by program nets and the MSIAM that are also parameterized, in this section we obtain an adequacy result in a parameterized setting. In other words, we are able to obtain a multi-token GoI interpretation of *any* specific language as long as the axioms of memory structures hold.

Definition 5.62 ($\mathbf{PCF}_{\mathrm{Mem}}$)**.** Given a memory structure $\mathrm{Mem} = (\mathrm{Mem}, \mathcal{I}, \mathcal{L})$, the language $\mathbf{PCF}_{\mathrm{Mem}}$ is defined by the following BNF:

$$\begin{aligned} \mathrm{M}, \mathrm{N}, \mathrm{P} ::= {} & x \mid \lambda x.\mathrm{MN} \mid \mathtt{let}\ \langle x, y\rangle = \mathrm{M}\ \mathtt{in}\ \mathrm{N} \mid \langle \mathrm{M}, \mathrm{N}\rangle \mid \\ & \mathtt{letrec}\ f\ x = \mathrm{M}\ \mathtt{in}\ \mathrm{N} \mid \\ & \mathtt{new} \mid \ell \mid \mathtt{if}\ \mathrm{P}\ \mathtt{then}\ \mathrm{M}\ \mathtt{else}\ \mathrm{N}, \\ A, B \quad ::= {} & \alpha \mid A \to B \mid A \times B \mid\ !A \end{aligned}$$

where $\ell \in \mathcal{L}$.

$$\frac{}{!\,\Delta \vdash \mathsf{new} : \alpha} \quad \frac{}{!\,\Delta, x : !(A \to B) \vdash x : A \to B} \quad \frac{A \text{ linear}}{!\,\Delta, x : A \vdash x : A}$$

$$\frac{!\,\Delta \vdash V : A \to B \quad V \text{ value}}{!\,\Delta \vdash V : !(A \to B)} \quad \frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x.M : A \to B}$$

$$\frac{!\,\Delta, \Gamma_1 \vdash M : A \to B \quad !\,\Delta, \Gamma_2 \vdash N : A}{!\,\Delta, \Gamma_1, \Gamma_2 \vdash MN : B}$$

$$\frac{!\,\Delta, \Gamma_1 \vdash M : A \quad !\,\Delta, \Gamma_2 \vdash N : B}{!\,\Delta, \Gamma_1, \Gamma_2 \vdash \langle M, N \rangle : A \times B}$$

$$\frac{!\,\Delta, \Gamma_1 \vdash M : A \times B \quad !\,\Delta, \Gamma_2, x : A, y : B \vdash N : C}{!\,\Delta, \Gamma_1, \Gamma_2 \vdash \mathtt{let}\, \langle x, y \rangle = M \,\mathtt{in}\, N : C}$$

$$\frac{\Delta \vdash P : \alpha \quad \cdot \vdash M : A \quad \cdot \vdash N : A}{\Delta \vdash \mathtt{if}\, P \,\mathtt{then}\, M \,\mathtt{else}\, N : A} \quad \frac{\mathrm{arity}(\ell) = n}{!\,\Delta \vdash \ell : \alpha^{\times n} \to \alpha^{\times n}}$$

$$\frac{!\,\Delta, f : !(A \to B), x : A \vdash M : B \quad !\,\Delta, \Gamma, f : !(A \to B) \vdash N : C}{!\,\Delta, \Gamma \vdash \mathtt{letrec}\, f\, x = M \,\mathtt{in}\, N : C}$$

Figure 5.15: Typing Rules.

Definition 5.63 (Typing Rules of $\mathbf{PCF}_{\mathrm{Mem}}$)**.** The typing rule of $\mathbf{PCF}_{\mathrm{Mem}}$ is as shown in Figure 5.15.

Once again, a status during evaluation of a term in $\mathbf{PCF}_{\mathrm{Mem}}$, called an *evaluation closure*, is given by a triple consisting of a term, a memory, and an indexing (partial) function. The reduction rules are as shown in Figure 5.16, where the syntax of *values* is the same as the usual one with operations ℓ added: $U, V ::= x \mid \lambda x.M \mid \langle U, V \rangle \mid \ell$. Here we choose the call-by-value evaluation strategy since it is the standard one in presence of the kinds of effects with which a memory structure deals, while the call-by-name can also be accommodated exactly in the same way as Chapter 4.

Definition 5.64 (Raw Evaluation Closure)**.** A *raw evaluation closure* of $\mathbf{PCF}_{\mathrm{Mem}}$ is a triple $(\mathrm{M}, \mathrm{ind}_{\mathrm{M}}, \mathrm{m}_{\mathrm{M}})$ where M is a term, $\mathrm{ind}_{\mathrm{M}}$ is an injective partial function from the set of free variables in M to $\mathcal{I}$, and $\mathrm{m} \in \mathrm{Mem}$.

Definition 5.65 (Evaluation Closure)**.** An *evaluation closure* of $\mathbf{PCF}_{\mathrm{Mem}}$ is an equivalence class over raw evaluation closures quotiented by permutations $\sigma \in \mathrm{FinBij}(\mathcal{I})$.

The translation of $\mathbf{PCF}_{\mathrm{Mem}}$ judgments into program nets is shown in Figure 5.17 and 5.18.

Remark 5.66. Precisely speaking, the syntax of $\mathbf{PCF}_{\mathrm{Mem}_{\mathbb{N}}}$ and PCF in Chapter 4 do not exactly coincide, because in $\mathbf{PCF}_{\mathrm{Mem}_{\mathbb{N}}}$ the successor and the predecessor are represented by function symbols of type $\mathbb{N} \to \mathbb{N}$ while they are constructors rather than function symbols in PCF. However they are essentially equivalent: the function symbol $\mathtt{succ}$ in $\mathbf{PCF}_{\mathrm{Mem}_{\mathbb{N}}}$ can be represented by $\lambda x.\mathtt{succ}\, x$ in PCF, and the construction $\mathtt{succ}\, \mathrm{M}$ in PCF can be represented by the function application $\mathtt{succ}\, \mathrm{M}$ in $\mathbf{PCF}_{\mathrm{Mem}_{\mathbb{N}}}$; moreover their normal forms are the same, both as terms and as nets that interpret them.

Theorem 5.67 (Adequacy between $\mathbf{PCF}_{\mathrm{Mem}}$ and Program Nets)**.** *Let* $\vdash \mathbf{M} : \alpha$. *Then,* $\mathbf{M}\Downarrow_p$ *if and only if* $\mathbf{M}^\dagger\Downarrow_p$.

$$(C[\mathsf{new}], \mathrm{ind}, \mathrm{m}) \to_{\mathrm{link}} (C[x], \mathrm{ind} \cup \{x \mapsto i\}, \mathrm{m})$$

$$(C[\ell\,\langle x_1, \ldots, x_n\rangle], \mathrm{ind}, \mathrm{m}) \to_{\mathrm{update}(\ell)} (C[\langle \vec{x}\rangle], \mathrm{ind}, \mathrm{update}(\vec{i}, \ell, \mathrm{m}))$$

$$(C[\mathtt{if}\, x\, \mathtt{then}\, M_{\mathtt{true}}\, \mathtt{else}\, M_{\mathtt{false}}], \mathrm{ind}, \mathrm{m})$$
$$\to_{\mathrm{test}(i)} \mathrm{test}(i, \mathrm{m})[\mathtt{true} := (M_{\mathtt{true}}, \mathrm{ind} \setminus \{x \mapsto i\}), \mathtt{false} := (M_{\mathtt{false}}, \mathrm{ind} \setminus \{x \mapsto i\})]$$

$$(C[(\lambda x.M)U], \mathrm{ind}, \mathrm{m}) \to (C[M\{x := U\}], \mathrm{ind}, \mathrm{m})$$

$$(C[\mathtt{let}\, \langle x, y\rangle = \langle U, V\rangle\, \mathtt{in}\, M], \mathrm{ind}, \mathrm{m}) \to (C[N[x := U, y := V]], \mathrm{ind}, \mathrm{m})$$

$$(C[\mathtt{letrec}\, f\, x = M\, \mathtt{in}\, N], \mathrm{ind}, \mathrm{m})$$
$$\to (C[N\{f := \lambda x.\mathtt{letrec}\, f\, x = M\, \mathtt{in}\, M\}], \mathrm{ind}, \mathrm{m})$$

Figure 5.16: Call-by-Value evaluation Rules for $\mathbf{PCF}_{\mathrm{Mem}}$

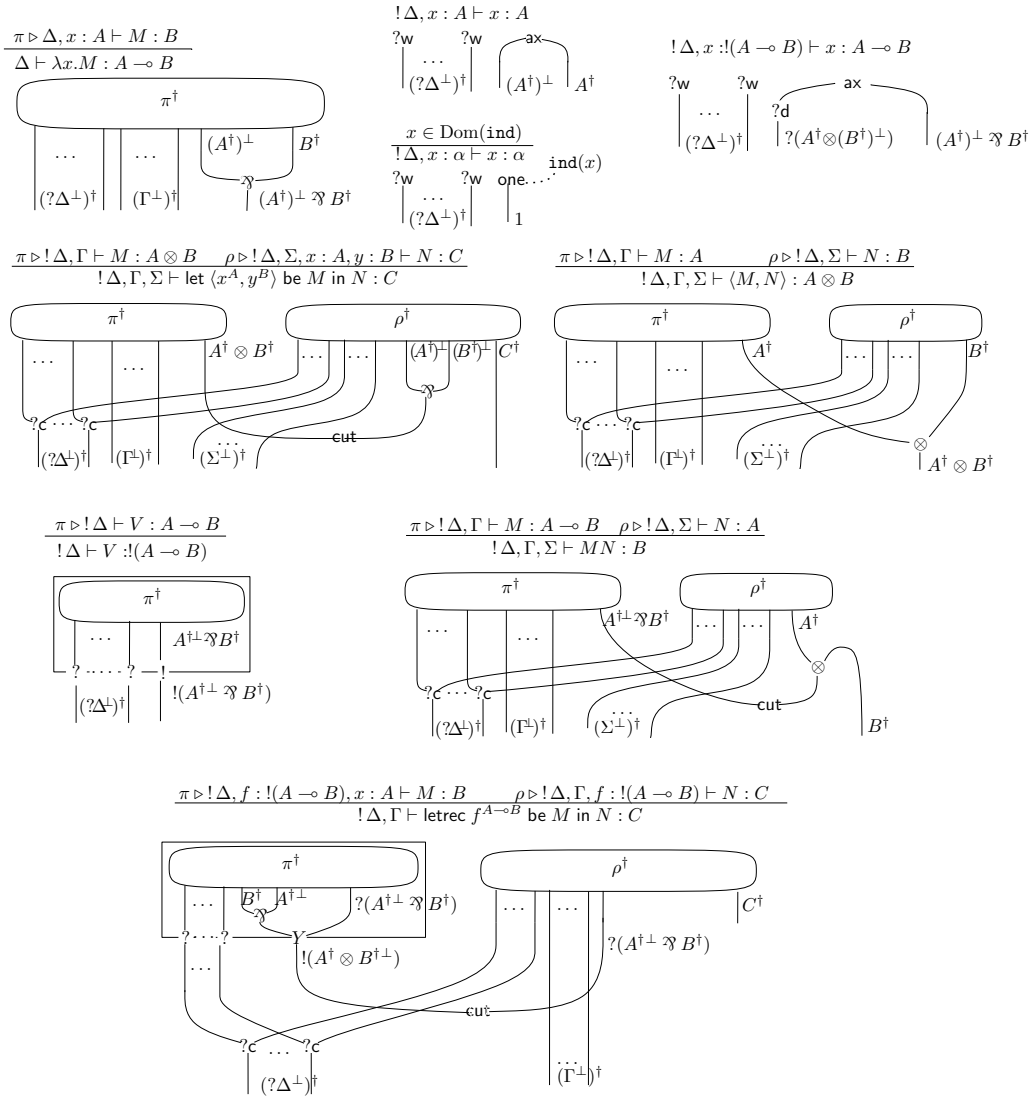


Figure 5.17: Call-by-Value Translation of $\mathbf{PCF}_{\mathrm{Mem}}$ (1)

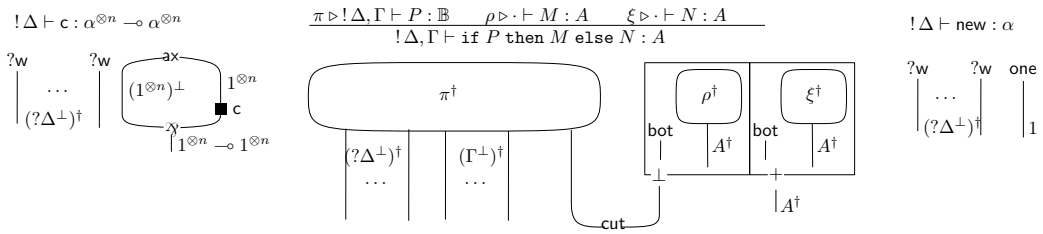


Figure 5.18: Call-by-Value Translation of $\mathbf{PCF}_{\mathrm{Mem}}$ (2)

The proof of Theorem 5.67 relies on some lemmas on the translation $(-)^\dagger$ below.

Lemma 5.68. *Let* $\mathbf{M} = (\mathrm{M}, \mathrm{ind}, \mathrm{m})$ *be an evaluation closure with* $\cdot \vdash \mathbf{M} \colon \alpha$*, and* μ *be a distribution of evaluation closures. We have:*

1. $\mathbf{M}$ *is in normal form (as an evaluation closure) if and only if* $\mathbf{M}^\dagger$ *is in normal form (as a program net).*
2. $\mathcal{T}(\mu) = \mathcal{T}(\mu^\dagger)$.

Proof. Item 1. can be checked by case analysis on evaluation rules (of $\mathbf{PCF}_{\mathrm{Mem}}$) and reduction rules (of program nets). Item 2. immediately follows from 1. □

Lemma 5.69. *Let* $\mathbf{M} = (\mathrm{M}, \mathrm{ind}, \mathrm{m})$ *be an evaluation closure with* $\cdot \vdash \mathbf{M} \colon \alpha$*. We have:*

1. *if* $\mathbf{M} \to \mu$*, then* $\mathbf{M}^\dagger \rightrightarrows^+ \mu^\dagger$.
2. *if* $\mu \rightrightarrows^* \nu$*, then* $\mu^\dagger \rightrightarrows^* \nu^\dagger$.

Proof. Again item 1. follows by case analysis on evaluation rules. Item 2. is a consequence of 1. □

Corollary 5.70. *Let* $\mathbf{M} = (\mathrm{M}, \mathrm{ind}, \mathrm{m})$ *be an evaluation closure with* $\cdot \vdash \mathbf{M} \colon \alpha$*, and* μ *be a distribution of evaluation closures. We have:*

1. *If* $\mathbf{M}^\dagger \rightsquigarrow \rho$*, then there exists* μ *satisfying* $\mathbf{M} \to \mu$ *with* $\mathbf{M}^\dagger \neq \mu^\dagger$.
2. *If* $\mathbf{M}^\dagger \rightrightarrows^k \rho$*, then there exists* μ *satisfying* $\mathbf{M} \rightrightarrows^* \mu$ *and* $\mathbf{M}^\dagger \rightrightarrows^m \mu^\dagger$ *for some* $m \geq k$.

Proof. (1.) Immediate consequence of Lemma 5.68 and 5.69 (2.) By induction. □

Now the proof proceeds as follows.

Proof of Theorem 5.67. Assume $\mathbf{M} \Downarrow_{p_{\mathrm{term}}}$ and $\mathbf{M}^\dagger \Downarrow_{p_{\mathrm{net}}}$; we prove $p_{\mathrm{term}} = p_{\mathrm{net}}$ by showing $p_{\mathrm{term}} \leq p_{\mathrm{net}}$ and $p_{\mathrm{term}} \geq p_{\mathrm{net}}$.
$\mathbf{p_{term} \leq p_{net}}$. Assume $\mathbf{M} \rightrightarrows^* \mu$ with $\mathcal{T}(\mu) = p_{\mathrm{term}}$, then $\mathbf{M}^\dagger \rightrightarrows^* \mu^\dagger$ (by Lemma 5.69.2) and $\mathcal{T}(\mu^\dagger) = p_{\mathrm{term}}$ (by Lemma 5.68.2). Thus $\mathbf{p_{term} \leq p_{net}}$ (because convergence is defined by supremum: see Definition 5.8).
$\mathbf{p_{term} \geq p_{net}}$. We prove that if $\mathbf{M}^\dagger \rightrightarrows^* \rho$ then there exists μ satisfying $\mathbf{M} \rightrightarrows^* \mu$ and $\mathcal{T}(\mu) \geq \mathcal{T}(\rho)$. Assume $\mathbf{M}^\dagger \rightrightarrows^* \rho$ is in k steps, i.e. $\mathbf{M}^\dagger \rightrightarrows^k \rho$. By Corollary 5.70, $\mathbf{M} \rightrightarrows^* \mu$ and $\mathbf{M}^\dagger \rightrightarrows^m \mu^\dagger$, with $m \geq k$. By Uniqueness of Normal Forms (Theorem 5.12.1) we have that $\mathcal{T}(\mu^\dagger) \geq \mathcal{T}(\rho)$. By Lemma 5.68, $\mathcal{T}(\mu) = \mathcal{T}(\mu^\dagger)$, from which we deduce the statement. □

Chapter 6

Conclusion and Future Work

We conclude the thesis reviewing the aims we stated in Chapter 1 (Section 6.1), and show several directions of future work (Section 6.2).

6.1 Conclusion

In the thesis we introduced comprehensive definitions of multi-token Geometry of Interaction machines and its underlying proof nets, and examined how the multi-token machine framework can adequately interpret a class of PCF-like languages possibly with choice effects. The three constructs we introduced (Fig. and Fig. 4.9), especially the sync node, are non-standard ones; nevertheless, the proof nets, called SMEYLL proof nets, are shown to have desirable properties (Cor. 4.9, Thm. 4.19, Cor. 4.31). The progress property, and the cut-elimination property as its consequence, are guaranteed by a correctness criterion as in linear logic, while the proofs get much more complicated. Then we in turn proved that the multi-token machine on SMEYLL nets (called the synchronous interaction abstract machine) also had desirable properties (Thm. 4.60, Thm. 4.49, Thm. 4.56). Deadlocks are by definition absent in the standard case; it is however proved by relying on the correctness criterion and by transferring properties of nets and the machine to each other, which is done in the standard case. The nets and machine are then applied to interpretation of PCF-like calculus, and the adequacy property is also shown. The fact that our token machine can distinguish the call-by-name and call-by-value translations suggests that the notion of paths calculated in our machine has some fundamental meaning despite its non-standard definition. Chapter 5 further extended the use of multi-token machine as a computational model. Using the notion of probabilistic abstract rewriting system that we introduced in this work, we uniformly constructed models of calculi parameterized by the notion of memory structures. A quantum calculus is interpreted as an instance of the model; soundness and adequacy are simply instantiated from parameterized soundness and adequacy theorems. This explains that the syntactic constraint seen in previous work [24, 45] is inessential, hence achieves the main aim of this thesis, *a better understanding of characters of dialogue-based semantics*. The technical details and difficulties we overcame in this thesis provide us a fairly firm basis for the forthcoming discussion on the two long-term goals, *implicit parallelism from a semantic viewpoint* and *uniform account for type systems in concurrent computation*, in future work. We give details on them as well as some other possible future directions of work relevant to the multi-token framework below.

6.2 Future Work

6.2.1 Implicit Parallelism

The relation between our work and implicit parallelism is left unexamined in the thesis. The multi-token character of the machine does suggest a form of parallelism inherent to programs: parallel movement of dereliction tokens implies that the correspondent function calls are independent of each other and thus can be evaluated in parallel, and the same argument holds for tokens generated from the unit nodes. However, the cost of path computation is not the same as the usual one; the standard single-token machine is known to be efficient (or, at least not very inefficient) according to existing work [37, 83], but the addition of synchronization and global conditions to generate tokens can be an obstacle in a possible implementation. The benefit of employing implicit parallelism comes from the fact that those parallelized evaluations do not need to be arbitrated, so it is probable that an implementation of the multi-token machine contributes to speed-up. Therefore, a possible contribution for this area is to use our framework not for execution of a program itself but for an analysis of the program. This might look similar to an approach based on runtime profiles [96], but since the multi-token machine is confluent we do not have to iterate the analysis and search for heuristics. Another possible advantage over existing approaches is that the parallelism of our framework is applicable not only to a pure functional language but also to language with probabilistic effects. The very argument that explains the availability of implicit parallelism is based on referential transparency. Thus, in general, it is not valid for languages with effects; our approach suggests that in some type of effects the argument still holds and possibly parallelized in an automatic way. As mentioned in Chapter 2, interpretation of our work from the viewpoint of PELCR [81] may also lead to a better understanding as well as an extraction of implementation. All this is, however, only speculative at this stage and requires further study.

6.2.2 Concurrency

The most promising (and perhaps the most interesting) direction of future work is application of our formulation of the multi-token machines to the modeling of *concurrent computation*. The idea itself is straightforward since the token machines defined in the thesis obviously carry a concurrent flavor. However, a suitable adaptation is required for the purpose. This is because proof nets do not exhibit a concurrent behavior in the sense that the reductions/transitions satisfy confluence (in the usual sense or the probabilistic sense). A hopeful solution is to consider some other graph-based concurrent system and utilize intuition and techniques of the multi-token systems we developed so far. This is indeed not a mere idea but ongoing work: we have carried out work [20][1] to obtain multi-token machine semantics of *multiport interaction net* [7, 70] that is a general and expressive graph reduction system capable of expressing concurrent computation. Moreover, we are trying to extend this line of research to obtain token machine semantics of *differential interaction net* [25] with Ugo Dal Lago and Damiano Mazza, which is ongoing work. Ultimately, we envisage to (re)construct behavioral type systems for some process calculus via our concurrent GoI framework, utilizing the fact that a GoI semantics carries certain information on the types

[1] Since this topic has been done as another student's master thesis topic, we do not include it as a part of this thesis.

with which the underlying structures are equipped.

6.2.3 Dialogue-Based Semantics and Quantum Computation

We successfully constructed an adequate semantics for a higher-order quantum programming language without the syntactic constraint seen in existing work [24, 45]. The fact that it is proved to be possible to give an adequate semantics with the multi-token machine framework, and that the two standard evaluation strategies can be treated uniformly, support our non-standard approach. This suggests that game semantics in Delbecque's work [24] might also be modified to one that is sound (and hopefully fully abstract) with respect to a quantum programming language without the constraint, on which we desired to obtain some clue through the work. Currently, we are unsure whether our multi-token machine framework leads to a concrete contribution to the area of quantum computation. However, the research on semantics of quantum programming languages as well as their application to practical implementation is still active (e.g. [80]); at this stage, exploration of as many alternative semantics as possible is valuable to find out the best suitable one for practical quantum computers in the approaching future.

6.2.4 Logical Basis

Another possible direction is to seek a formal logic for the proof nets we have defined. The system of SMEYLL nets is surely novel, but how novel they are could be more clearly presented by relating the systems with some existent ones. The new constructs we added, in particular sync nodes, look peculiar as logical operations; currently we have not found any logical equivalent of them except for Montelatici's original work on Y box [75]. Such a formulation would lead to import existing notions from traditional logics and export the notions we invented to those. On this topic, the logical system presented in [47] also looks relevant to SMEYLL proof nets, though the connection is still to be clarified. If we succeed to identify the connection, SMEYLL will be formally related to Abelian logic, which possibly yields further investigation on the role of sync nodes for example.

We have also been trying to characterize the multi-token machine in the language of category theory. A candidate is to use multiset monad for describing multiple tokens running around the nets. We expect that not only our multi-token machine but also a broad class of concurrent computation would be explained in a single categorical framework if this can spell out the behavior of our machine. More precise comparison with [45] would also be possible, and in that case we can possibly merge two successful extensions of the Geometry of Interaction approach.

6.2.5 Compiler Construction

One notable benefit of using Geometry of Interaction as programming language semantics is that it often provides us a direct implementation of a compiler for the language we interpret with GoI. Concerning this perspective, the multi-token machine we introduced in the thesis does not seem suitable for this purpose. Advantages to derive such a compiler from GoI semantics are: (a) simplicity of the machine, (b) space efficiency, (c) time efficiency, (d) correctness of the derived compiler by construction. (b) is most probably lost in our machine since we now have to record the global state during execution to correctly assure that exactly one token is generated for each unit/dereliction node in each copy of exponential boxes. (c) is also doubtful for the same reason: we have to check the conditions on those copies frequently, which should correspond to a sharp slow down regarding

execution time. (a) and (d) among these reasons survive for our machine, but without the efficiency merits it would not be of interest to users.

Still, there is at least theoretically interesting way to relate our multi-token machine to compiler construction. The work by Fredriksson and Ghica [32] constructs a compiler for *distributed computation* from Geometry of Interaction semantics. Although the languages we interpreted with our multi-token machine in the thesis are all sequential, the machine apparently carries a distributed flavor. Whether we can realize our multi-token machine as a physically distributed system is itself of interest, and possibly some efficiency can be achieved, too.

6.2.6 Extension to Broader Effects

The probabilistic branchings we studied in the thesis and non-deterministic branching can be unified by some categorical axioms [46]. However, our axiomatization of memory structure does not accommodate non-deterministic branching (without probabilities). Another desirable effect to be unified is global states with update and lookup. The reason why it is not contained in our current axiomatization is that multiple updates of global state break the diamond property as a (probabilistic) rewriting system, thus the behavior deviates from the branching effects we dealt in the thesis. Relaxing the requirement on memory possibly makes sense and broader kinds of effects like the above two would be in the scope of our framework; this is surely future work.

Index

References

[1] *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 1989.

[2] Samson Abramsky. Information, processes and games. In Johan van Benthem and Pieter Adriaans, editors, *Philosophy of Information*, pages 483–549, 2008.

[3] Samson Abramsky and Bob Coecke. Physical traces: Quantum vs. classical information processing. *Electr. Notes Theor. Comput. Sci.*, 69:1–22, 2002.

[4] Samson Abramsky, Esfandiar Haghverdi, and Philip J. Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002.

[5] Samson Abramsky and Radha Jagadeesan. New foundations for the geometry of interaction. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992*, pages 211–222. IEEE Computer Society, 1992.

[6] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.

[7] Vladimir Alexiev. *Non-deterministic interaction nets*. PhD thesis, University of Alberta, 1999.

[8] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[9] Andrea Asperti and Giovanna M. Dore. Yet another correctness criterion for multiplicative linear logic with MIX. In Anil Nerode and Yuri Matiyasevich, editors, *Logical Foundations of Computer Science, Third International Symposium, LFCS'94, St. Petersburg, Russia, July 11-14, 1994, Proceedings*, volume 813 of *Lecture Notes in Computer Science*, pages 34–46. Springer, 1994.

[10] Marc Bagnol, Amina Doumane, and Alexis Saurin. On the dependencies of logical rules. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2015.

[11] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. Psi-calculi: Mobile processes, nominal data, and logic. In *Proceedings of the*

24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA, pages 39–48. IEEE Computer Society, 2009.

[12] Charles H Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. In *IEEE International Conference on Computers, Systems and Signal Processing*, volume 175, page 8, 1984.

[13] Daniil Berezun and Neil D. Jones. Compiling untyped lambda calculus to lower-level code by game semantics and partial evaluation (invited paper). In Ulrik Pagh Schultz and Jeremy Yallop, editors, *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2017, Paris, France, January 18-20, 2017*, pages 1–11. ACM, 2017.

[14] M. Bezem and J. W. Klop. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*, chapter Abstract Reduction Systems. Cambridge University Press, 2003.

[15] Giuseppe Castagna and Andrew D. Gordon, editors. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 2017.

[16] Kevin K. H. Cheung and Michele Mosca. Decomposing finite abelian groups. *Quantum Information & Computation*, 1(3):26–32, 2001.

[17] U. Dal Lago and M. Zorzi. Wave-style token machines and quantum lambda calculi. In *LINEARITY*, pages 64–78, 2014.

[18] Ugo Dal Lago, Claudia Faggian, Benoît Valiron, and Akira Yoshimizu. Parallelism and synchronization in an infinitary context. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 559–572. IEEE Computer Society, 2015.

[19] Ugo Dal Lago, Claudia Faggian, Benoît Valiron, and Akira Yoshimizu. The geometry of parallelism: classical, probabilistic, and quantum effects. In Castagna and Gordon [15], pages 833–845.

[20] Ugo Dal Lago, Ryo Tanaka, and Akira Yoshimizu. The geometry of concurrent interaction: Handling multiple ports by way of multiple tokens. To appear in proceedings of LICS 2017.

[21] Vincent Danos and Russell Harmer. Probabilistic game semantics. *ACM Trans. Comput. Log.*, 3(3):359–382, 2002.

[22] Vincent Danos and Laurent Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28(3):181–203, 1989.

[23] Vincent Danos and Laurent Regnier. Reversible, irreversible and optimal lambda-machines. *Electr. Notes Theor. Comput. Sci.*, 3:40–60, 1996.

[24] Yannick Delbecque. Game semantics for quantum data. *Electr. Notes Theor. Comput. Sci.*, 270(1):41–57, 2011.

[25] Thomas Ehrhard and Laurent Regnier. Differential interaction nets. *Electr. Notes Theor. Comput. Sci.*, 123:35–74, 2005.

[26] Thomas Ehrhard, Christine Tasson, and Michele Pagani. Probabilistic coherence spaces are fully abstract for probabilistic PCF. In Jagannathan and Sewell [55], pages 309–320.

[27] Ebbe Elsborg. *Bigraphs: Modelling, Simulation, and Type Systems. On Bigraphs for Ubiquitous Computing and on Bigraphical Type Systems.* PhD thesis, IT-Universitetet i København, 2009.

[28] Ebbe Elsborg, Thomas T. Hildebrandt, and Davide Sangiorgi. Type systems for bigraphs. In Christos Kaklamanis and Flemming Nielson, editors, *Trustworthy Global Computing, 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*, volume 5474 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2008.

[29] Uffe Engberg and Glynn Winskel. Petri nets as models of linear logic. In André Arnold, editor, *CAAP '90, 15th Colloquium on Trees in Algebra and Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, volume 431 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 1990.

[30] Javier Esparza. Decidability and complexity of petri net problems - an introduction. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. Springer, 1996.

[31] Javier Esparza and Mogens Nielsen. Decidability issues for petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.

[32] Olle Fredriksson and Dan R. Ghica. Seamless distributed computing from the geometry of interaction. In Catuscia Palamidessi and Mark Dermot Ryan, editors, *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers*, volume 8191 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2012.

[33] Simon J. Gay and Rajagopal Nagarajan. Communicating quantum processes. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 145–157. ACM, 2005.

[34] Vijay Ghelot. *A Proof-Theoretic Approach to Semantics of Concurrency.* PhD thesis, University of Pennsylvania, 1992.

[35] Dan R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 363–375. ACM, 2007.

[36] Dan R. Ghica and Alex Smith. Geometry of synthesis iii: resource management through type inference. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 345–356. ACM, 2011.

[37] Dan R. Ghica, Alex Smith, and Satnam Singh. Geometry of synthesis iv: compiling affine recursion into static hardware. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *ICFP*, pages 221–233. ACM, 2011.

[38] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[39] Jean-Yves Girard. Geometry of interaction 1: Interpretation of system F. *Logic Colloquium 88*, 1989.

[40] Jean-Yves Girard. Towards a geometry of interaction. *Contemporary Mathematics*, 92, 1989.

[41] Sergey Goncharov and Lutz Schröder. A relatively complete generic hoare logic for order-enriched effects. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 273–282. IEEE Computer Society, 2013.

[42] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In Ravi Sethi, editor, *POPL*, pages 15–26. ACM Press, 1992.

[43] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In Hans-Juergen Boehm and Cormac Flanagan, editors, *PLDI*, pages 333–342. ACM, 2013.

[44] Ramesh Hariharan and V. Vinay. String matching in õ(sqrt(n)+sqrt(m)) quantum time. *J. Discrete Algorithms*, 1(1):103–110, 2003.

[45] Ichiro Hasuo and Naohiko Hoshino. Semantics of higher-order quantum computation via geometry of interaction. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, pages 237–246. IEEE Computer Society, 2011.

[46] Ichiro Hasuo, Bart Jacobs, and Ana Sokolova. Generic trace semantics via coinduction. *Logical Methods in Computer Science*, 3(4), 2007.

[47] Yoichi Hirai. Session types in abelian logic. In Nobuko Yoshida and Wim Vanderbauwhede, editors, *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013.*, volume 137 of *EPTCS*, pages 33–52, 2013.

[48] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[49] Guido Hogen, Andrea Kindler, and Rita Loogen. Automatic parallelization of lazy functional programs. In Bernd Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 1992.

[50] Naohiko Hoshino. A modified goi interpretation for a linear functional programming language and its adequacy. In Martin Hofmann, editor,

Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings, volume 6604 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2011.

[51] Naohiko Hoshino, Koko Muroya, and Ichiro Hasuo. Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 52:1–52:10. ACM, 2014.

[52] Hans Hüttel. Typed ψ-calculi. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, volume 6901 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2011.

[53] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[54] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

[55] Suresh Jagannathan and Peter Sewell, editors. *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM, 2014.

[56] Kurt Jensen. Coloured petri nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer, 1986.

[57] C. Jones and Gordon D. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989* [1], pages 186–195.

[58] Neil D. Jones and David A. Schmidt. Compiler generation from denotational semantics. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark, January 14-18, 1980*, volume 94 of *Lecture Notes in Computer Science*, pages 70–93. Springer, 1980.

[59] Simon L. Peyton Jones. Parallel implementations of functional programming languages. *Comput. J.*, 32(2):175–186, 1989.

[60] Philippe Jorrand and Marie Lalire. From quantum physics to programming languages: A process algebraic approach. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms, International Workshop UPP 2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and*

Invited Papers, volume 3566 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004.

[61] Emmanuel Knill. Conventions for quantum pseudocode. Technical report, Los Alamos National Lab., NM (United States), 1996.

[62] Naoki Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*, pages 128–139. IEEE Computer Society, 1997.

[63] Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 205–225. Springer, 2010.

[64] Peter J Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[65] François Le Gall. An efficient quantum algorithm for some instances of the group isomorphism problem. In Jean-Yves Marion and Thomas Schwentick, editors, *27th International Symposium on Theoretical Aspects of Computer Science, STACS 2010, March 4-6, 2010, Nancy, France*, volume 5 of *LIPIcs*, pages 549–560. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[66] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 1999.

[67] Paul Lorenzen. Ein dialogisches konstruktivitätskriterium. Infinitistic Methods, op. cit., 1961.

[68] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In Jeanne Ferrante and P. Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 47–57. ACM Press, 1988.

[69] Ian Mackie. The geometry of interaction machine. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 198–208. ACM Press, 1995.

[70] Damiano Mazza. *Interaction Nets: Semantics and Concurrent Extensions*. PhD thesis, Université de la Méditerranée and Università degli Studi Roma Tre, 2006.

[71] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[72] Robin Milner. Bigraphical reactive systems. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, volume 2154 of *Lecture Notes in Computer Science*, pages 16–35. Springer, 2001.

[73] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[74] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989* [1], pages 14–23.

[75] Raphaël Montelatici. Polarized proof nets with cycles and fixpoints semantics. In Martin Hofmann, editor, *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings.*, volume 2701 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2003.

[76] Koko Muroya, Naohiko Hoshino, and Ichiro Hasuo. Memoryful geometry of interaction II: recursion and adequacy. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 748–760. ACM, 2016.

[77] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge Univ. Press, 2000.

[78] B. Ömer. Quantum programming in qcl. Master's thesis, Institute of Information Systems, Technical University of Vienna, 2000.

[79] Michele Pagani, Peter Selinger, and Benoît Valiron. Applying quantitative semantics to higher-order quantum computing. In Jagannathan and Sewell [55], pages 647–658.

[80] Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: a core language for quantum circuits. In Castagna and Gordon [15], pages 846–858.

[81] Marco Pedicini and Francesco Quaglia. PELCR: parallel environment for optimal lambda-calculus reduction. *ACM Trans. Comput. Log.*, 8(3):14, 2007.

[82] Carl Adam Petri. Communication with automata. Technical Report RADC-TR-65-377, Rome Air Dev. Center, New York, 1966. English translation of the original PhD thesis (1962).

[83] J. S. Pinto. *Implantation Parallèle avec la Logique Linéaire (Applications des Réseaux d'Interaction et de la Géométrie de l'Interaction)*. PhD thesis, École Polytechnique, 2001. Main text in English.

[84] Andrew M Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge University Press, 2013.

[85] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[86] N. Saheb-Djahromi. Probabilistic LCF. In Józef Winkowski, editor, *Mathematical Foundations of Computer Science 1978, Proceedings, 7th Symposium, Zakopane, Poland, September 4-8, 1978*, volume 64 of *Lecture Notes in Computer Science*, pages 442–451. Springer, 1978.

[87] Ulrich Schöpp. On interaction, continuations and defunctionalization. In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings*, volume 7941 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2013.

[88] Ulrich Schöpp. Call-by-value in a basic logic for interaction. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *Lecture Notes in Computer Science*, pages 428–448. Springer, 2014.

[89] Dana Scott. *Outline of a mathematical theory of computation.* Oxford University Computing Laboratory, Programming Research Group, 1970.

[90] Philip Scott. Tutorial on geometry of interaction., 2004. Tutorial talk at FMCS 2004. Slides available online at http://www.site.uottawa.ca/ phil/papers/GoI.tutorial.pdf.

[91] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[92] Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2005.

[93] Peter Selinger and Benoît Valiron. Quantum lambda calculus. In Simon Gay and Ian Mackie, editors, *Semantic Techniques in Quantum Computation*, pages 135–172. Cambridge Univ. Press, 2009.

[94] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[95] G. Tremblay and G. R. Gao. The impact of laziness on parallelism and the limits of strictness analysis. In *PROCEEDINGS HIGH PERFORMANCE FUNCTIONAL COMPUTING*, pages 119–133, 1995.

[96] José Manuel Calderón Trilla and Colin Runciman. Improving implicit parallelism. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 153–164. ACM, 2015.

[97] Dave Wecker and Krysta M. Svore. Liqui|>: A software design architecture and domain-specific language for quantum computing. *CoRR*, abs/1402.4467, 2014.

[98] Nobuko Yoshida. Graph types for monadic mobile processes. In Vijay Chandru and V. Vinay, editors, *Foundations of Software Technology and*

Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer, 1996.

[99] Akira Yoshimizu, Ichiro Hasuo, Claudia Faggian, and Ugo Dal Lago. Measurements in proof nets as higher-order quantum circuits. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 371–391. Springer, 2014.

[100] Ming-Yuan Zhu. Denotational semantics of programming languages and compiler generation in powerepsilon. *SIGPLAN Notices*, 36(9):39–53, 2001.