Hyper-Lambda Calculi
(ハイパーラムダ計算)

by

Yoichi Hirai
平井洋一

A Doctoral Thesis

Submitted to
the Graduate School of Information Science and Technology,
the University of Tokyo
on December 13, 2012
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Thesis Supervisor: Masami Hagiya　萩谷昌己
Professor of Computer Science

ABSTRACT

We propose hyper-lambda calculi, the typed lambda calculi based on hypersequent calculi. A hyper-lambda term is a finite sequence of lambda terms, which represent concurrent processes. We give three concrete hyper-lambda calculi: one synchronous and the other two asynchronous. All employ a pair of communication primitives exchanging their inputs. In the synchronous case, both sides succeed. In the asynchronous cases, at least one side obtains the other side's input. The synchronous calculus implements message-passing communication and session types; the asynchronous calculus characterizes shared-memory waitfree communication. Among processes of a typed hyper-lambda term, all succeed in the synchronous case while at least one succeeds in the asynchronous case. Logically, the processes are interpreted conjunctively in the synchronous case but disjunctively in the asynchronous case. The synchronous calculus is based on Abelian logic: $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$ on top of multiplicative additive fragment of intuitionistic linear logic (without some units); one of the asynchronous calculi is based on Gödel-Dummett logic: $(\varphi \supset \psi) \vee (\psi \supset \varphi)$ on top of intuitionistic logic. The hyper-lambda calculi are in Curry-Howard correspondence with the deduction systems for these logics. We also give a similar development for monoidal t-norm logic and implement the Gödel-Dummett case using Haskell.

ハイパーラムダ計算すなわちハイパー推件計算に基づく型付きラムダ計算を提案する．ハイパーラムダ項はラムダ項の有限列で，並列プロセスを表す．同期と非同期と二つのハイパーラムダ計算を与える．両者ともに，互いに入力を交換する通信子の対を用いる．同期ハイパーラムダ計算では，通信子双方が成功する．非同期ハイパーラムダ計算では，通信子の少なくとも片方がもう片方の入力を獲得する．同期ハイパーラムダ計算はメッセージパッシング通信とセッション型を実装し，非同期ハイパーラムダ計算は共有メモリの無待機計算を特徴づける．型付きハイパーラムダ項のプロセスどものうち，同期ハイパーラムダ計算では全てが成功するが，非同期ハイパーラムダ計算では少なくとも一つが成功する．論理的にプロセスどもは，同期ハイパーラムダ計算では連言で，非同期ハイパーラムダ計算では選言で解釈される．同期ハイパーラムダ計算はアーベル論理に基づく: 直観主義線形論理の乗法的加法的断片 (除いくつかの単位論理式) に公理 $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$ を加えた論理である．非同期ハイパーラムダ計算はゲーデル・ダメット論理に基づく: 直観主義論理に公理 $(\varphi \supset \psi) \vee (\psi \supset \varphi)$ を加えた論理である．これらのハイパーラムダ計算は，それぞれの論理の演繹体系とカリー・ハワード対応にある．Monoidal t-norm 論理に基づく変種と，Haskell を用いた実装も扱う．

Acknowledgements

The author thanks my adviser Masami Hagiya for his patience and some decisive comments at important moments. Even before composing this thesis, the author became indebted to many of the juries and their colleagues. In an MLG (mathematical logic group) meeting, Hiroakira Ono asked the author about the constructive content of Dummett axiom; an attempt to answer that question eventually became Chapter 3. Naoki Kobayashi's request for a stronger communication primitive resulted in Chapter 2. Ichiro Hasuo gave the author some chances of talks and indispensable advice on visiting the Netherlands and studying there. During the defense procedure, the author merited from interests and advice from Shinichi Honiden and Andrzej Murawski as well as from other juries.

The work about hyper-lambda calculus is encouraged by feedbacks from ACAN (Algebraic and Coalgebraic Approaches to Non-Classical Logics Workshop) and OPLSS'11 participants, Pisa Proof Theory Workshop, FLOPS-2012 and numerous other occasions.

Grant-in-Aid for JSPS Fellows 23-6978 supported my second and third years of PhD, of which almost one year was spent in ILPS, the University of Amsterdam. The author thanks Maarten Marx, Maarten de Rijke and other members of ILPS for comfortable environments and stimulating discussions during my stay. Especially, it was a unique training to work on heavy problems for months with Maarten Marx, Alessandro Facchini and Evgeny Sherkhonov.

The author thanks Tadeusz Litak for encouragements and information on relevant research. At the University of Tokyo, discussions with Tatsuya Abe, Yoshihiko Kakutani and Masahiro Hamano nurtured my taste on formalistic, semantic and graphical approaches.

Contents

List of Figures

List of Examples

List of Theorems

For definitions, see Index at the end of the thesis.

Chapter 1

Introduction

1.1 Our Contributions

We propose hyper-lambda calculi. Instead of lambda terms in ordinary lambda calculi, we use a finite sequence of lambda terms called hyper-lambda terms. We give two such examples: one models synchronous send-receive communication (Chapter 2) and the other models asynchronous waitfree read-write communication (Chapter 3). The two hyper-lambda calculi are in Curry-Howard correspondence with hypersequent formulation of logics: Abelian logic and Gödel-Dummett logic. Gödel-Dummett logic is an intermediate logic. An intermediate logic is a logic *between* classical and intuitionistic logics, where 'between' is defined using the set-theoretic inclusion of valid logical formulae of each logic. Abelian logic is a substructural logic.

Hosoi and Ono [82] declared that they chose to study intermediate logics in general rather than studying specific intermediate logics. However, concrete results about specific subjects matter when the results contain new phenomena. This thesis is intended to contain such concrete discoveries about specific intermediate and substructural logics.

The more famous logic among the two, Gödel-Dummett logic, is a typical intermediate logic known from 1950's. Our method is the Curry-Howard correspondence, also known from 1940's. Our result is characterization of waitfreedom, a concept in the theory of distributed computation extensively studied in 1990's. Since both sides of the correspondence are already known, the discovery is a replay of Curry's surprise.

After developing a hyper-lambda calculus for Abelian logic, we found that the resulting programming language is similar to a session type systems by Giunti and Vasconcelos [62] although they were not aware of the underlying logic. The independent discoveries constitute another instance of Curry–Howard correspondence "not by design".

We list our contributions from the most important:

1. developing a lambda calculus using a hypersequent calculus (Chapters 2, 3);
2. identifying the computational ability of Gödel-Dummett logic with waitfreedom (Chapter 3);
3. encoding session types using a seemingly unknown axiom $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$ (which we call the Amida axiom) on top of the multiplicative additive fragment of intuitionistic linear logic (Chapter 2),
4. discovery of a new proof system for Abelian logic (Chapter 2);
5. using conjunctive hypersequents for the first time (Chapter 2);
6. giving a typed lambda calculus for monoidal t-norm logic (Chapter 4); and
7. implementing a hyper-lambda calculus in Haskell (Chapter 5).

Our first contribution is the technique for our second contribution. We developed a lambda calculus based on Avron's hypersequents [8]. Avron [8] himself noted that it would be important to develop a lambda calculus for hypersequents. The lambda calculus relies on Avron's hypersequents [8]. A hypersequent calculus is a variant of the deduction system called sequent calculus. In sequent calculus, each step of a proof tree concludes a sequent $\Gamma \vdash \varphi$ that consists of a finite sequence of logical formulae Γ and a logical formula φ. The sequent $\Gamma \vdash \varphi$ is interpreted as an implication. In hypersequent calculus, each step of a proof tree concludes a hypersequent instead of a sequent. A *hypersequent* is a sequence of sequents delimited by $\mid$, e.g., $\Gamma \vdash \varphi \mid \Delta \vdash \psi \mid \cdots$. Also here, each component is interpreted as an implication, and then the whole hypersequent is interpreted as the disjunction of all those implications. When we interpret proofs as programs, we take the components as concurrent processes. Following the original disjunctive interpretation of components, we regard the proof tree as the guarantee of success of at least one process.

Our second contribution is the new approach of interpreting proofs in Gödel-Dummett logic as concurrently executable programs for waitfree computation. Although Avron [8] noticed his hypersequent calculus has something to do with concurrency (as the title of [8] contains the phrase "intermediate logics for concurrency"), it was unknown that the computational interpretation of Gödel-Dummett logic coincides with the degree of synchronization called waitfreedom. This discovery constitutes our second contribution.

According to Sørensen and Urzyczyn [130, p.97], the Curry-Howard isomorphism [130] was first made precise by Curry and Feys [37, **9**E and **9**F]. The intuitionistic propositional logic and the typed lambda calculus had been independently invented but Curry discovered them to be the same thing. The "double discovery" (Wadler [146]) is considered to affirm the importance of the typed lambda calculi. In this thesis we witness a replay of the "double discovery" with different casts: Gödel-Dummett logic and the waitfreedom. Both of these were born in the early eras of their respective academic disciplines: mathematical logic bore Gödel Dummett logic in 1950's [43] and the computer science bore waitfree computation in 1970's [93]. Chapter 3 is about the previously unknown connection between these two.

This is the first such computational interpretation for intermediate logics (Figure 1.1). Another significance of this contribution lies in giving interpretation of nondeterminism in typed lambda calculi as scheduling of concurrent processes. In the simply typed lambda calculus for intuitionistic propositional logic, all typed terms have a unique normal form. However, in typed lambda calculi for classical propositional logic, there can be multiple normal forms unless we employ an evaluation strategy or limit the set of reductions. The lack of unique normal forms in the classical propositional proofs has puzzled logicians for decades. The most famous example is Lafont's example [61, B.1.], which provides a way to equate any two proofs for the same formula (if we equate proofs before and after cut-elimination). In computer science, such a phenomenon is called nondeterminism caused by scheduling. Dynamic behavior involves time. One simple notion of time is that of totally ordered events where one event happens before the other or the other before the one. This sentence is syntactically similar to Dummett's axiom that states one proposition implies the other or the other implies the one. We investigate whether this syntactic similarity is reflected in the dynamic semantics of logics: namely, the lambda calculi.

Our third contribution is about encoding of session types into a linear type system. Although the approach is similar to that of Caires and Pfenning [26] and Wadler [146], the Amida calculus has an additional axiom so that it can type some processes that Pfenning or Wadler's type systems cannot.

Our fourth contribution is a side effect of the third contribution. The type system we developed for our third contribution is an unknown proof system for Abelian logic [27]. We moreover take the essence of the new proof system and obtain a kind of proof nets called the Amida nets. Syntactically, the correct proof structure can

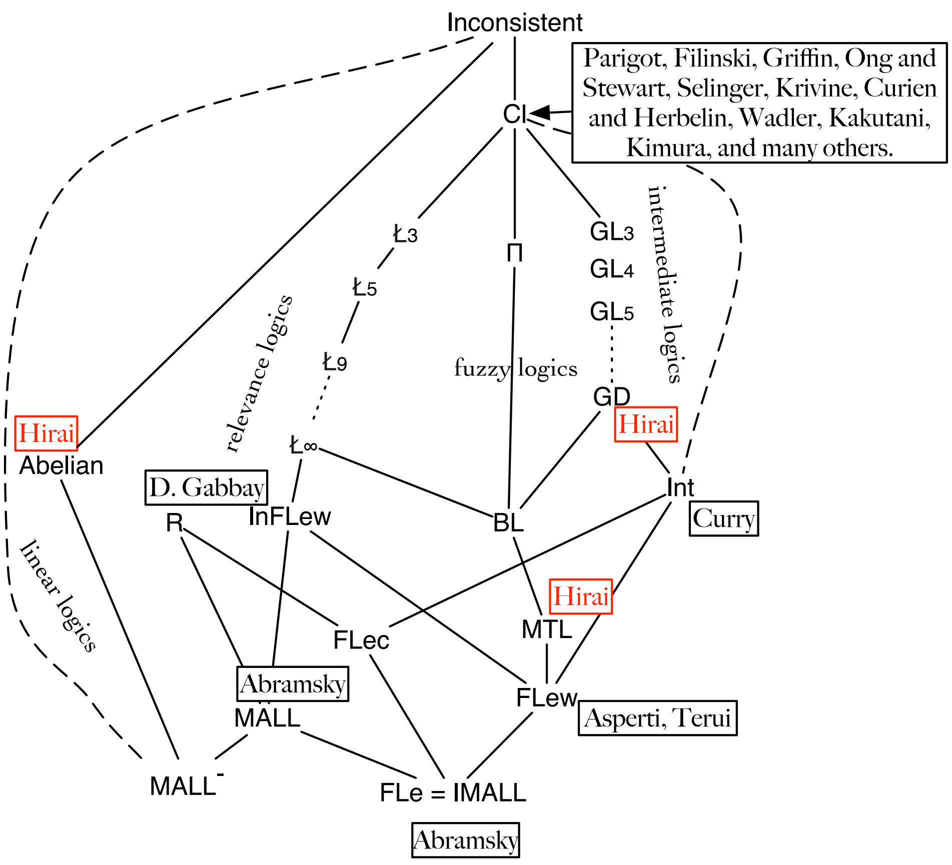


Figure 1.1: The substructural and intermediate logics for which lambda calculi are found. The underlying Hasse diagram of well-known substructural logics is taken from [57, p. 120] with slight modifications. The names in boxes refer to people who developed lambda calculi for these logics. GD stands for Gödel-Dummett logic, for which a lambda calculus will be given in Chapter 3. Abelian stands for Abelian logic [27, 103, 104] discussed in Chapter 2. MTL is monoidal t-norm logic [46], for which we give a hyper-lambda calculus in Chapter 4. FLe is for the full Lambek calculus with exchange rule [57, p.86], which is also known as the intuitionistic multiplicative additive fragment of linear logic (IMALL). MALL stands for its classical version. For these fragments of linear logic, Abramsky [3] gave lambda calculi. MALL^- [59] is a fragment of MALL without additive units. FLew has weakening on top of FLe. FLew is also known as intuitionistic affine logic. Affine logics lack contraction, which causes exponential size increase during cut-elimination process. Asperti gave light affine logic [7]. Terui [132] gave an affine typed lambda calculus for polynomial time computation. R stands for relevance logic [139], for which Gabbay and de Queiroz [55] gave a lambda calculus. Int stands for the intuitionistic propositional logic. The original Curry-Howard isomorphism was found for this logic by Curry [36]. Cl stands for the classical propositional logic. There is intensive research going on for the computational interpretation of classical logic [35, 49, 66, 85, 115, 117, 127, 144, 145], of which Daisuke Kimura's thesis [87] provides an overview. Inconsistent stands for the logic of all logical formulae.

be characterized with the Amida edges, which is similar to the structure of Amida lottery. The previous chapters gave the computational interpretations of disjunctive formulae like $(\varphi \supset \psi) \vee (\psi \supset \varphi)$ or $(\varphi \multimap \psi) \oplus (\psi \multimap \varphi)$. In this chapter, we try replacing these disjunctions with conjunctions. In the former case, the change renders the logic inconsistent; if we add the axiom $(\varphi \supset \psi) \wedge (\psi \supset \varphi)$ to the intuitionistic propositional logic, we can prove any formula. However in the latter case, the change does not make the system meaningless. In Chapter 2, we introduce the axioms of the form $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$ on top of IMALL, intuitionistic multiplicative additive linear logic. In essence, the axiom allows two processes to wait for one another and then exchange information.

Our fifth contribution is the use of conjunctive hypersequents. Hypersequents have been around since Avron [8], but in all cases, different components in a hypersequent were interpreted disjunctively. In our formalization of Abelian logic in Chapter 2, we use conjunctive hypersequents, where different components are interpreted conjunctively. This is the first application of such conjunctive hypersequents.

Our sixth contribution is a typed lambda calculus for monoidal t-norm logic. Monoidal t-norm logic (MTL) is the affine version of Gödel-Dummett logic. MTL is considered by some to be the weakest fuzzy logic. Following our method of interpreting Gödel-Dummett logic proofs as waitfree protocols, we interpret the MTL proofs in our deduction system as asynchronously communicating processes. There we employ the second-order formulation and perform parametricity arguments.

Our seventh contribution is implementing a hyper-lambda calculus in Haskell. By implementing a hyper-lambda calculus for waitfreedom, we obtain a empirical justification of our second contribution.

The content of Chapter 3 appears in a conference paper by the author [77] although we have applied substantial modifications since then.

1.2 Preliminaries

We use "iff" as an abbreviation for "if and only if." We use set-theoretic concepts such as sets, relations and functions. For a set X, 2^X denotes the powerset of X. $X \setminus Y$ denotes the subset of X that consists of the elements of X that are not elements of Y. Everywhere, we consider a countably infinite set of logical formulae. A logic is a set of logical formulae, thus, there are at most $2^{\mathbb{N}}$ different logics once the set of logical formulae is fixed. Likewise, we consider at most countably many proofs and lambda terms.

We denote partial maps as graphs. For example, $\{0 \mapsto M, 1 \mapsto N\}$ denotes a partial map that maps 0 to M and 1 to N. We denote by $\mathrm{dom}(f)$ the domain of f, that is, the set of elements that are mapped to something. We also denote a partial map as a sequence whose index set is the domain. For example, $(y_i)_{i \in I}$ denotes the partial map that maps $i \in I$ to y_i. For partial maps whose domains are disjoint, we use $\sqcup$ to denote the union. In other words, when f and g have disjoint domains, $f \sqcup g$ denotes the partial map that maps x to $f(x)$ or $g(x)$ when one of these is defined, or otherwise to nothing.

We use BNF (Backus Naur Form) for giving inductive definitions for sets of finite sequences of symbols.

Example 1.2.1 (An example of BNF) *When we say we define formulae φ by BNF:*

$$\varphi ::= \bot \mid p \mid (\varphi \supset \varphi)$$

where p is a propositional variable, we actually define a set Φ which is the smallest set such that

- *$\bot$ is in Φ*
- *each propositional variable is in Φ and*
- *if φ and ψ are in Φ, then $(\varphi \supset \psi)$ is in Φ;*

and then we declare that we call elements of Φ formulae.

We take binary operators of the symbol $\supset$ or $\multimap$, to be right associative. For example, $\varphi \multimap \psi \multimap \theta$ is interpreted as $\varphi \multimap (\psi \multimap \theta)$.

1.3 History

We briefly review the history of mathematical logic and computer science to the extent relevant to this thesis. Especially, we focus on the treatments of Dummett's axiom $(a \supset b) \vee (b \supset a)$ using the historical techniques. We confine ourselves to the developments of propositional logic and ignore anything related to predicate logic or formalization of mathematics.

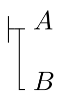
1.3.1 Birth of Formal Logic

Implication is an important concept. One treatment of implications is called the *material implication*. The material implication can be traced back at least to Frege's

Begriffsschrift [54] in 1879. There, in the section called "conditionality," he begins by establishing four cases [54, p. 13]:

1. A is affirmed and B is affirmed;
2. A is affirmed and B is denied;
3. A is denied and B is affirmed;
4. A is denied and B is denied.

Then he defines a notation involving A and B



which "stands for the judgment that *the third of these possibilities does not take place, but one of the three other does*[1]" [54, p. 14]. In the contemporary common mathematical terminology, this is equivalent to stating B implies A. It was Bertrand Russell who called this implication the material implication, according to Edgington [44]. After Brouwer's notation, we still use $\vdash$ for judgments in our formal systems. The view of logical formulae as graphical objects will be inherited to proof nets (Section 2.6).

1.3.2 Intuitionistic Propositional Logic

The intuitionistic propositional logic is a logic, that is, a set of logical formulae. Although the name "intuitionistic" comes from Brouwer's intuitionism, the original claims of intuitionism are unrelated to this thesis. Brouwer was skeptical about the value of formalization of mathematics and considered the studies of formal axiomatized logic as "mathematics of the second and third order" [141, p. 10]. Nonetheless, Brouwer approved publication of his student Arend Heyting's work on defining a set of logical formulae as the intuitionistic propositional logic.

In 1930, Heyting [74] developed a deduction system for the intuitionistic propositional logic. The presentation is similar to the contemporary one except some notational differences. The logical connectives $\{\wedge, \vee, \supset, \neg\}$ are the same as today, the first three binary and the last unary[2]. Logical formulae are constructed using these connectives and the propositional variables. Although he sometimes used parentheses, he still used more points around outer connectives (e.g. $b \supset\cdot\, a \supset b$ indicates the left $\supset$

[1]The emphasis is found in the English translation [54, p. 14].

[2]Though, in this thesis, we prefer having nullary $\bot$ as a primitive and define $\neg a$ as an abbreviation for $a \supset \bot$.

is outer and the right one is inner). Instead we use parentheses. These formulae are axioms[3], i.e. assumed to be "correct formulae" [74]:

2.1. $a \supset (a \wedge a)$

2.11. $(a \wedge b) \supset (b \wedge a)$

2.12. $(a \supset b) \supset (a \wedge c) \supset (b \wedge c)$

2.13. $((a \supset b) \wedge (b \supset c)) \supset (a \supset c)$

2.14. $b \supset (a \supset b)$

2.15. $(a \wedge (a \supset b)) \supset b$

3.1. $a \supset (a \vee b)$

3.11. $(a \vee b) \supset (b \vee a)$

3.12. $((a \supset c) \wedge (b \supset c)) \supset ((a \vee b) \supset c)$

4.1. $\neg a \supset (a \supset b)$

4.11. $((a \supset b) \wedge (a \supset \neg b)) \supset \neg a$.

There are more formulae given by the following rules [74]:

1.2. If a and b are correct formulas[4], then $a \wedge b$ is a correct formula.

1.3. If a and $a \supset b$ are correct formulas, then b is a correct formula.

Following the intuitionists' belief that "it is in principle impossible to set up a system of formulas that would be equivalent to intuitionistic mathematics, for the possibilities of thought cannot be reduced to a finite number of rules set up in advance" [74], Heyting did not pursue completeness results for the deduction system except a remark mentioning Glivenko's theorem [63, 64]. In other words, he did not pursue arguing that his system is strong enough. On the contrary, he argued his system is not too strong. He used what would be called algebraic semantics today in order to show that each axiom is independent from the other axioms and that the excluded middle is unprovable. For example, in order to refute the excluded middle, he used the following tables [74]:

[3]The axioms with numberings are taken from Heyting [74].

[4]In quotes, we do not adjust the plural "formulas" and "formulae."

$\supset$	0	1	2
0	0	0	0
1	1	0	1
2	2	0	0

$\wedge$	0	1	2
0	0	1	2
1	1	1	1
2	2	1	2

$\vee$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	2

$\neg$	0	1	2
	1	0	1

.

When we assign one of $\{0, 1, 2\}$ to each propositional variable, we can extend the assignment to all formulae using these tables. For binary operators, the columns stand for the prearguments (the arguments on the left of the operator) and the rows stand for the postarguments (the arguments on the right of the operator). For example, if we assign 1 to a and 2 to b, $a \supset b$ obtains value 0. Under any assignment to propositional variables, all of Heyting's axioms are assigned 0. Moreover, these tables have two desirable properties:

1. $0 \wedge 0 = 0$
2. $0 \supset a$ has the value 0 only when $a = 0$.

From these, all "correct formulae" in Heyting's deduction system receive the value 0 under any assignments to propositional variables. However, if we assign 2 to a, $(\neg\neg a) \supset a$ is assigned 2. Thus, we can conclude that $\neg\neg a \supset a$ is not a correct formula in Heyting's deduction system. Note that the natural order between natural numbers plays no role here: it is not relevant that 2 is larger than 1 or 0 is less than 1 (in other places he uses "all positive and negative whole numbers and 0"). Although Heyting used natural numbers here, he already obtained an equivalent notion of what is called Heyting algebra today[5].

We can refute Dummett's axiom $(a \supset b) \vee (b \supset a)$ in the same method. Indeed, according to the tables below, when we assign 1 to a and 2 to b, $(a \supset b) \vee (b \supset a)$ obtains 4, which is not 0. Since all correct formulae receive 0, $(a \supset b) \vee (b \supset a)$ is not a correct formula.

$\supset$	0	1	2	3	4
0	0	0	0	0	0
1	1	0	1	0	1
2	2	2	0	0	2
3	3	3	3	0	3
4	4	0	0	0	0

$\wedge$	0	1	2	3	4
0	0	1	2	3	4
1	1	1	3	3	1
2	2	3	2	3	2
3	3	3	3	3	3
4	4	1	2	3	4

$\vee$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	4	1	4
2	0	4	2	2	4
3	0	1	2	3	4
4	0	4	4	4	4

$\neg$	0	1	2	3	4
	3	3	3	0	3

[5] Although he did not explicitly say that he can define a partial order using the semantics for $\supset$.

Gödel

In 1932, Gödel published a short note [65] on intuitionistic propositional logic, where he proved two theorems: that the intuitionistic propositional logic cannot be seen as a many-valued logic and that there are infinitely many propositional logics between the intuitionistic propositional logic and the "ordinary" (in today's terminology, classical) propositional logic. The second result is the first contribution to the realm of intermediate logics (according to Troelstra [47, p. 223]). For distinguishing those intermediate logics and the intuitionistic propositional logic, he uses a formula F_n for each positive natural number n:

$$F_n = \bigvee_{1 \leq i < k \leq n} ((a_i \supset a_k) \wedge (a_k \supset a_i)) \quad .$$

On an n-element chain, F_{n+1} is valid while F_n might not be satisfied. As a result, no F_n is valid in the intuitionistic propositional logic.

Among the formulae F_n, especially, the formula F_2, which is $(a_1 \supset a_2) \wedge (a_2 \supset a_1)$, brings contradiction into classical or intuitionistic logic. Moreover, any logic (closed under modus ponens and substitution) having F_2 as a theorem is inconsistent. In Chapter 2 we investigate a logic with an axiom similar to F_2: $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$, where $\otimes$ is the multiplicative conjunction and $\multimap$ is the linear implication. That logic is consistent because it lacks weakening and contraction rules. Such attempt has been made possible by the development of linear and substructural logics explained below in 1.3.9.

In the final sentence, Gödel [65] states what is known as disjunction property today[6]: "besides, the following holds with full generality: a formula of the form $A \vee B$ can only be provable in H if either A or B is provable in H," where H is Heyting's calculus. According to this statement, it is obvious that Dummett's axiom $(p \supset q) \vee (q \supset p)$ is not provable.

Although Troelstra [47, p. 223] writes "the reasons for studying intermediate logics are mainly technical," we find that one typical intermediate logic, Gödel-Dummett logic, has a computational interpretation that has already been known: waitfreedom. The connection between Gödel-Dummett logic and waitfreedom will be treated in Chapter 3.

[6]Disjunction property can be proved by soundness and completeness with respect to Kripke models. There is another, syntactic technique called Aczel's slash [136, Ch. 3. 5.7.].

1.3.3 The Brouwer-Heyting-Kolmogorov Interpretation of Logical Connectives

Kolmogorov's view of implication as problem reduction

Thanks to Heyting [74], we have a formal characterization of the intuitionistic implication. In 1932, Kolmogorov [91] introduced a "calculus of problems" that coincides with the intuitionistic propositional logic. In the calculus of problems, the logical connectives $\vee, \wedge, \supset$ connect problems together to form another problem.

> If a and b are two problems, then $a \wedge b$ designates the problem "to solve both problems a and b," while $a \vee b$ designates the problem "to solve at least one of the problems a and b." Furthermore, $a \supset b$ is the problem "to solve b provided that the solution for a is given" or, equivalently, "to reduce the solution of b to the solution of a." ... $\neg a$ designates the problem "to obtain a contradiction provided that the solution of a is given." [91, p. 329]

Here, the intuitionistic implication is explained as reduction of problems. He continues to validate Heyting [74]'s axioms, and the deduction rules with respect to the problem calculus interpretation.

Furthermore, he finds a reason for not including the law of excluded middle $a \vee \neg a$ by saying "one must possess a general method either to prove or to reduce to a contradiction any proposition. If our reader does not consider himself to be omniscient, he will probably determine that the formula cannot be found on the list of problems solved by him" [91]. This argument is enough to reject Dummett's axiom $(a \supset b) \vee (b \supset a)$ because one must possess a general method, given any two problems, to determine whether one problem can be reduced to the other or the other way around. For the computational meaning of the law of excluded middle, we have to wait until 1990's. And the computational interpretation of Dummett's axiom is presented in Chapter 3 in this thesis.

Realization as Typed Lambda Terms

One formulation of the Brouwer-Heyting-Kolmogorov (BHK) interpretation reads: "a proof of the implication $\varphi \supset \psi$ is a construction which permits us to transform any proof of φ into a proof of ψ" [136, Ch. 1, 3.1.]. The BHK interpretation does not specify what is a proof or what kind of transformation witnesses implication.

1.3.4 Natural Deduction and Sequent Calculus

Gentzen's Deduction Systems

The deduction system of Heyting [74] characterizes what is still called intuitionistic propositional logic today. However, Heyting's system has one drawback, which is shared with almost all[7] other Hilbert-style derivation systems: in order to prove a logical formula, sometimes we have to mention a larger, more complicated formula. Actually, there are alternative formulations of intuitionistic (and classical) propositional logic by Gentzen [58] where we only have to mention subformulae of the formula being proven. The desirable property is called the subformula property. The two proof systems are called natural deduction and sequent calculus.

In contrast to the system of Heyting [74] where a proof yields a finite set of "correct formulae," a proof in natural deduction and sequent calculus yields a pair of assumptions and conclusions. Gentzen [58] used the form

$$\mathbf{A_1}, \ldots, \mathbf{A}_{\boldsymbol{\mu}} \longrightarrow \mathbf{B}$$

as a sequent. We will use $\vdash$ instead of $\longrightarrow$. In any case, a sequent stands for the implication: the conjunction of $\mathbf{A_i}$'s implies $\mathbf{B}$.

In natural deduction, the assumptions and conclusions are presented vertically. For example,

$$\frac{A \wedge B}{B}$$

has a single assumption $A \wedge B$ and a conclusion B. The same content can be shown as

$$\frac{\overline{A \wedge B \vdash A \wedge B}}{A \wedge B \vdash B} \quad .$$

The horizontal line shows an application of an inference rule[8]. The sequent on top of the inference rule is called the *assumption* of the rule and the sequent below is called the *conclusion* of the rule. Natural deduction has *introduction rules* and *elimination rules* for each logical connectives. An introduction rule of a connective contains the connective in the conclusion but not in the assumptions (if any). An elimination rule of a connective contains the connective in one of the assumptions but not in the conclusion. For example, the introduction and elimination rules for $\wedge$ (conjunction[9]) is as follows:

[7]Even Hilbert-style deduction systems can avoid this drawback if all theorems are axioms.

[8]In [58], an inference rule is called an inference figure schema.

[9]Gentzen [58] and Prawitz [119] used &.

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}\,.$$

These rules are the meaning of conjunction $\wedge$. In sequent calculus, the introduction rules stay the same but called the right rules because they operate on the right hand side of sequents. The elimination rules are rewritten so that they operate on the left side of sequents and they are called the left rules.

In Chapter 2, we will use a hypersequent calculus for presenting Abelian logic. In Chapter 3, we will use hypersequent-style natural deduction for presenting Gödel-Dummett logic.

Prawitz's Analysis of Natural Deduction

Gentzen [58] did not prove the subformula property immediately for natural deduction. He proved the subformula property for sequent calculus first and then after that obtained the property of natural deduction as a corollary. On the other hand, Prawitz [119] studied natural deduction in itself.

Inversion principle states that an occurrence of a logical connective in a proof can be removed when the connective is introduced by an introduction rule, and then immediately below, eliminated by an elimination rule. A logical formula occurrence containing such a connective is called a maximal formula. For example, for conjunction, a natural deduction derivation [119, p. 36] can be reduced

$$\text{from } \wedge\mathcal{E}\,\frac{\wedge\mathcal{I}\,\dfrac{\dfrac{\Sigma_0}{A} \qquad \dfrac{\Sigma_1}{B}}{A \wedge B}}{A} \text{ to } \frac{\Sigma_0}{A}$$

where Σ_0 and Σ_1 stand for derivations with conclusions A and B respectively. In the reduction, the occurrence of $\wedge$ is removed. The same holds for other logical connectives; thus we can remove maximal formulae.

Further, Prawitz [119, Chapter IV] formulated a weaker notion called maximal segments. Above, in the definition of maximal formulae, the elimination rule must be placed immediately below the corresponding introduction rule. Prawitz allowed other inference lines between the introduction and elimination rules and defined maximal segments[10]. A derivation without maximal segments is called normal. Prawitz proved the existence of a normal derivation of $\Gamma \vdash A$ given any derivation of $\Gamma \vdash A$.

[10] A maximal segment can contain many occurrences of the same formula without being influenced by any inference rules. Such "syntactic bureaucracy" can be removed by proof nets. See Section 2.6 for an example.

Prawitz [119] treated first-order, second-order and modal logics but we do not elaborate.

The reductions and the normal form coincide with the β-reduction and the normal form in the typed lambda calculi under Curry-Howard isomorphism.

Prior's Tonk

Prior [120] invented a logical connective called tonk.

$$\frac{\varphi}{\varphi \text{ tonk } \psi} \qquad \frac{\psi}{\varphi \text{ tonk } \psi} \qquad \frac{\varphi \text{ tonk } \psi}{\varphi} \qquad \frac{\varphi \text{ tonk } \psi}{\psi}$$

From this, inversion principle requires this reduction:

$$\text{from } \cfrac{\cfrac{\varphi}{\varphi \text{ tonk } \psi}}{\psi} \text{ to } \frac{\varphi}{\psi} \quad .$$

The reduct is considered nonsense because if there are any theorems all formulae must be theorems. This is an argument refuting the tonk operator. However, in Chapter 2, we pursue the possibility of compensating the nonsense by the dual nonsense, namely:

$$\frac{\varphi \qquad \psi}{\psi \qquad \varphi} \quad .$$

Prior's paper is titled "the runabout inference-ticket." In Chapter 2, we are going to consider the round-trip inference-ticket.

1.3.5 Curry-Howard Isomorphism

At the core of computer science lies the interplay of static formalism and dynamic behavior. We can find examples in typed lambda calculi, where static formalism of type derivations interacts with dynamic behavior of lambda terms. Type derivations are static objects associating lambda terms to types. The reduction relation on terms gives dynamics, defining which term reduces to which. Static type derivations can guarantee dynamic properties of programs such as strong normalization [61] and more specific properties using parametricity arguments [124].

The Curry-Howard isomorphism is originally a correspondence between intuitionistic propositional logic proofs and typed lambda terms, but the name combination Curry-Howard has obtained a more general meaning spanning over the correspondence between proofs and programs in general. The phrase "computational interpretation" is most often used for the Curry-Howard isomorphism [3, 18, 99, 118]. Sørensen and Urzyczyn [130] recently presented a comprehensive reference book on the topic, containing lots of historical and bibliographical remarks.

Curry's Discovery

The Curry-Howard isomorphism is originally the correspondence of the typed lambda terms and the proofs of the implicational fragment[11] of the intuitionistic propositional logic. According to [130], the first explicit statement of the correspondence appears in the retiring presidential address to the Association for Symbolic Logic titled "the combinatory foundations of mathematical logic" [36]. The footnote 28 of [36] reads:

> Note the similarity of the postulates for F and those for P. If in any of the former postulates we change F to P and drop the combinator we have the corresponding postulate for P

where a postulate for F is something like $\vdash FXYf$ which represents the statement that f belongs to the class of functions from X to Y; and P is the implicational fragment of the intuitionistic propositional logic. On the same page, there is Postulate (PC):

$$\vdash (\beta \supset (\alpha \supset \gamma)) \supset (\alpha \supset (\beta \supset \gamma))$$

and the corresponding Postulate (FC):

$$\vdash F(F\beta(F\alpha\gamma))(F\alpha(F\beta\gamma))C$$

where C is $\lambda^3 xyz \cdot xzy$ and the notation $F\alpha\beta$ shows the functional character of a function from α to β. The postulate (FC) is said to state the functional character of C. In this thesis, we use sequent style natural deduction system so that these postulates can be derived as

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x\!:\!\beta \supset (\alpha \supset \gamma) \vdash x\!:\!\beta \supset (\alpha \supset \gamma)} \quad \dfrac{}{z\!:\!\beta \vdash z\!:\!\beta}}{x\!:\!\beta \supset (\alpha \supset \gamma), z\!:\!\beta \vdash xz\!:\!\alpha \supset \gamma} \quad \dfrac{}{y\!:\!\alpha \vdash y\!:\!\alpha}}{x\!:\!\beta \supset (\alpha \supset \gamma), z\!:\!\beta, y\!:\!\alpha \vdash xzy\!:\!\gamma}}{x\!:\!\beta \supset (\alpha \supset \gamma), y\!:\!\alpha \vdash \lambda z.xzy\!:\!\beta \supset \gamma}}{x\!:\!\beta \supset (\alpha \supset \gamma) \vdash \lambda y.\lambda z.xzy\!:\!\alpha \supset (\beta \supset \gamma)}}{\vdash \lambda x.\lambda y.\lambda z.xzy\!:\!(\beta \supset (\alpha \supset \gamma)) \supset (\alpha \supset (\beta \supset \gamma))} \ .$$

Indeed, the last sequent associates the term $\lambda x.\lambda y.\lambda z.xzy$, which is the combinator C, to the logical formula $(\beta \supset (\alpha \supset \gamma)) \supset (\alpha \supset (\beta \supset \gamma))$. Throughout this thesis, we use colons $:$ to combine terms with types.

Henceforth, we do not give deduction systems of a logic and typing rules for a typed lambda calculi separately because the former can be obtained from the latter by this

[11]The implicational fragment of a logic can be defined by taking formulae only containing implications but no other connectives.

correspondence. The precise statements for the correspondence appear in Curry and Feys [37, 9E], a section titled “analogies with propositional algebra.”

Curry-Howard isomorphism provides one realization of BHK interpretation: proofs as lambda terms where the introduction rule of implication is realized as the lambda abstraction and the elimination rule of implication is realized as the application of lambda terms. This encoding has a desirable property: the reductions in the lambda calculus correspond to the reductions of proofs that remove detours. For example, suppose a natural deduction proof introduces an implication and then immediately eliminates the implication. This proof can be encoded as a lambda term whose outermost structure is a β-redex: $(\lambda x.M)N$. The result of the β-reduction $M[N/x]$ encodes a proof tree using the same assumptions as the original and concluding the same formula as the original, yet without the aforementioned detour. A proof of implication allows transformation of proofs by means of substitution. The encoding of proofs as lambda terms is traditionally called the Curry-Howard isomorphism.

The material implications are justified in classical propositional logic. The latter view on implication, provided by BHK-interpretation, is most naturally embodied in the situation of intuitionistic propositional logic. The last century saw their generalization called intermediate logics [137] (or superintuitionistic logics), of which a typical example is Gödel-Dummett logic. In this thesis we investigate the Curry-Howard isomorphism for Gödel-Dummett logic. The Gödel-Dummett logic validates formulae of the form $(\varphi \supset \psi) \vee (\psi \supset \varphi)$, which is known as Dummett's axiom. We add a construct to the simply typed lambda calculus that witnesses Dummett's axiom. After the intermediate logics, the generalization went further to substructural logics [57], which contains all the intermediate logics as well as the (intuitionistic) multiplicative additive fragment of linear logic. In the later chapters, we make a typing system that lacks contraction and weakening in Chapters 2 and 4.

Application to Programming Languages

Landin [94] noticed some similarities between ALGOL 60 syntax and the untyped lambda calculus. Since then, many programming languages came out of the Curry-Howard isomorphism: most notably so-called the ML family languages like Standard ML [108], OCaml [109], SML# [114], Haskell [98], F# [105] and so on.

However, none has employed Dummett axiom $(\varphi \supset \psi) \vee (\psi \supset \varphi)$ or the Amida axiom[12] $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$ as a type for a language primitive.

[12]For explanation of $\otimes$ and $\multimap$, see 1.3.9.

1.3.6 Gödel-Dummett Logic

Dummett's Axiomatization

Dummett [43] considers a semantics on $\{0, 1, 2, \ldots, \omega\}$ where $\wedge$ is interpreted as maximum, $\vee$ as minimum, $\bot$ as ω and implication as a function $\hat{\supset}$ where[13]

$$x \hat{\supset} y = \begin{cases} 0 & \text{if } x \geq y \\ y & \text{if } x < y \end{cases} .$$

He axiomatized the logic by adding $(p \supset q) \vee (q \supset p)$ on top of the axioms for the intuitionistic propositional logic. Thomas [133] axiomatized the logic on n-element chains using Dummett's axiom and the formula F_{n+1} that appeared in Gödel [65], which is rewritten using only implications but no disjunction or conjunction.

Sonobe's Gentzen-Style Calculus

Sonobe [129] presented a sequent calculus for Gödel-Dummett logic and proved cut-elimination theorem for it. On top of intuitionistic propositional logic, there is only one rule. However, the number of assumptions of the rule is not constantly bounded. The rule can have assumptions as many as any natural number. After looking at a hyper-lambda calculus λ-GD in Chapter 3, we can translate a proof in Sonobe's system into a λ-GD typing derivation and then find the computational content of the original proof. However, the author has never seen an attempt of determining the computational interpretation of Sonobe's cut-elimination.

Avron's Hypersequents

Avron [8] invented the hypersequent calculus. A *hypersequent* is a finite sequence of sequents:

$$\Gamma_0 \vdash \varphi_0 \mid \Gamma_1 \vdash \varphi_1 \mid \cdots \mid \Gamma_n \vdash \varphi_n \quad (n \geq 0) .$$

We use metavariable $\mathcal{H}$ for a hypersequent. For Gödel-Dummett logic, Avron [8] formulated the communication rule

$$\frac{\mathcal{H}_0 \mid \Gamma_0', \Gamma_0 \vdash \varphi_0 \qquad \mathcal{H}_1 \mid \Gamma_1', \Gamma_1 \vdash \varphi_1}{\mathcal{H}_0 \mid \mathcal{H}_1 \mid \Gamma_1', \Gamma_0 \vdash \varphi_0 \mid \Gamma_0', \Gamma_1 \vdash \varphi_1} .$$

The rule contains no logical connectives. That means in order to apply the rule, we do not have to pattern-match formulae except comparing whether two formulae are identical or not. Such a rule is called a *structural rule* as opposed to a logical rule.

[13]To be precise, Dummett did not use absurdity $\bot$ but negation $\neg$ although they are interdefinable in existence of implication $\supset$ and conjunction $\wedge$.

For the hypersequent calculus with the *communication rule*, Avron [8] showed cut-elimination theorem. He asked what is the computational content of Gödel-Dummett and other intermediate logics (see the beginning of Chapter 3 for quotation).

1.3.7 Waitfreedom

A waitfree protocol over shared memory [71] assigns a program to each process within the restriction that no process waits for another process. Some tasks can be solved by a well-chosen waitfree protocol while the others cannot. For example, it is waitfreely impossible for each processes to attain the input values of the other processes. On the other hand, it is waitfreely possible for at least one of the processes to attain the input values of the other processes.

Herlihy and Shavit [72] characterized waitfree computation using simplicial topology. Using their characterization, Gafni and Koutsoupias [56] showed that it is undecidable whether a task is waitfreely solvable or not.

1.3.8 Intermediate (Superintuitionistic) Logics

In this thesis, a *logic* is a set of logical formulae that is closed under substitution and modus ponens[14]. Elements of a logic are called *theorems*. The *substitution*-closedness means that if φ is a theorem, $\varphi[\psi/X]$ (i.e. the logical formula obtained by replacing all occurrences of X with ψ) is also a theorem. By *modus ponens* closedness, if $\varphi \supset \psi$ (or $\varphi \multimap \psi$) and φ are theorems, ψ is also a theorem. An intermediate logic is a consistent logic that contains the intuitionistic propositional logic. A logic is *consistent* when it does not contain all logical formulae.

Early intermediate logicians seem to have focused on algebraic aspects on intermediate logics in general. Troelstra [47, p. 223] writes "the reasons for studying intermediate logics are mainly technical." In a survey paper, Hosoi and Ono [82] say:

> The study of intermediate logics seems to have two aspects: One is to study particularities of a certain logic, and the other is to take the intermediate logics as a whole and to study the relations between the logics or some structures recognized in that system. We think that an intermediate logic is simply an algebraic system bearing some structural resemblance to the *logic* in the usual sense and that it is not a *logic* on which some kind of mathematics can or must be constructed. So, the second approach seems to

[14]There are, however, logics not closed for substitution such as dynamic epistemic logic [140] and inquisitive logic [32]. Thus in general it would be fair to say that a logic is a set of logical formulae.

be reasonable for us, and we have been mostly working with the intension of grasping the algebraic structure of the whole system of the intermediate logic[15].

Their approach has been successful. One spectacular result by Maksimova [96] states that there are only seven consistent superintuitionistic logics with the Craig interpolation property. The Craig interpolation property holds for a logic L iff $\alpha \supset \beta \in L$ implies existence of a formula χ such that $\alpha \supset \chi, \chi \supset \beta \in L$ and χ only contains propositional variables that appear in both α and β. The seven logics include the intuitionistic propositional logic, the logic of the weak excluded middle, Gödel-Dummett logic and classical logic. Although the Craig interpolation property is useful for program verification [45, 100, 138], the above result itself is only of theoretical interest.

We take the first approach described by Hosoi and Ono: namely, studying particular logics instead of intermediate logics in general. This thesis is about particular logics: Gödel-Dummett logic and Abelian logic. We claim that these logics can be used for developing programming languages. The Curry-Howard isomorphism justifies our investigation of these particular logics. Gödel-Dummett logic is one typical intermediate logic. Abelian logic, on the other hand, is not an intermediate logic because it lacks structural rules called weakening and contraction. Such logics are called substructural logics, whose history will be followed in 1.3.9.

We have to note that there have been attempts to apply some intermediate logics for mathematical reasoning, especially the truth theory. Hájek et al. [68] tried to use Łukasiewicz logic in order to resolve the liar's paradox. The idea is to assign the truth value 0.5 to the sentence "this sentence is false." When a sentence has the truth value x, a sentence that claims falsehood of the first sentence should have the truth value $1 - x$. When the first and second sentences are identical, it should have the truth value 0.5. Using Łukasiewicz logic, Hájek et al. [68] worked in Peano arithmetic and found a consistent formulation (in the sense there exists a model to each finite subtheory). However, in the case of axiomatic set theory with comprehension, Hájek [67] showed that having induction over natural numbers is contradictory[16].

Computationally inclined research on intermediate logics can be found in proof searching. The most typical intermediate logic, Gödel-Dummett logic has many proof searching implementations. Currently, the fastest solver the author is aware of is Fiorino's EPDL [50]. Fiorino benchmarked his and other implementations on a problem set for intuitionistic logic called ILTP library [121]. Related to proof searching,

[15]Emphases by the authors of the original paper.

[16]Yatabe [148] found an easier proof of this.

Fermüller [48] gave a game semantics for Gödel-Dummett logic, which is based on Lorenzen game [130] and essentially proof searching bottom-to-up. The game contains concurrent subgames but he gave no explicit mentions on waitfree computation.

On the other approach of proof reduction, there have been no lambda calculi or programming languages based on intermediate logics, despite the fact cut-elimination results are obtained by Sonobe [129] and Avron [8].

1.3.9 Substructural Logics

In sequent calculus or the sequent style natural deduction, when we apply most rules, we need to make pattern matching on the logical formulae. For example, in order to apply this $\wedge$L rule of sequent calculus for classical logic,

$$\wedge\mathrm{L}\ \frac{\Gamma, \varphi, \psi \vdash \Delta}{\Gamma, \varphi \wedge \psi \vdash \Delta}$$

we have to find a formula on the left hand side sequent whose top connective is $\wedge$.

Today, the logics without some structural rules are called *substructural logics*, the smallest of which is called the full Lambek calculus (FL). There have been intensive studies on substructural logics mainly from the algebraic approach [57].

Throughout this thesis, we consider only logics with the structural rule called exchange. The exchange rule allows permutation of formulae in contexts. The well-known substructural logics with the exchange rule are shown in Figure 1.1.

BCI and BCK Logics

When we add exchange rule to FL, we obtain BCI logic. Further, when we add weakening to BCI logic, we obtain BCK logic [116]. *BCK logic* has three axioms named after well-known combinators. Indeed, the logic can be characterized as the smallest set closed under modus ponens and substitution that contains

(B) $(\varphi \supset \psi) \supset ((\chi \supset \varphi) \supset (\chi \supset \psi))$

(C) $(\varphi \supset (\psi \supset \chi)) \supset (\psi \supset (\varphi \supset \chi))$ and

(K) $\varphi \supset (\psi \supset \varphi)$.

From these, by modus ponens and substitution, (I) $\varphi \supset \varphi$ follows, but (W) $(\psi \supset (\psi \supset \varphi)) \supset (\psi \supset \varphi)$ does not follow[17]. Thus, the left hand side contraction is not admissible in the sequent calculus for the BCK logic. A rule

[17]Here, the capital alphabet W is a name of a combinator. Elsewhere, W stands for a structural rule weakening.

$$\frac{\mathcal{S}}{\mathcal{S}'}$$

is *admissible* iff $\mathcal{E}'$ is derivable whenever $\mathcal{E}$ is.

Linear Logics

Girard found the linear logic [59]. To explain this logic, let us present two formulations of the $\wedge$-right rule in the sequent calculus of intuitionistic propositional logic:

$$\frac{\Gamma \vdash \varphi \qquad \Delta \vdash \psi}{\Gamma, \Delta \vdash \varphi \wedge \psi} \qquad\qquad \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}$$

Using one of these rule, the other can be defined as a macro (an abbreviation for a longer construction) because, from bottom to top, we can copy (*contraction*) or remove (*weakening*) formulae in contexts. However if we do not allow such structural rules, the two different formulations characterize different conjunctions. The left one is called *multiplicative* conjunction and the right one is called *additive* conjunction.

$$\frac{\Gamma \vdash \varphi \qquad \Delta \vdash \psi}{\Gamma, \Delta \vdash \varphi \otimes \psi} \qquad\qquad \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \mathbin{\&} \psi}$$

In general, a binary operator is called multiplicative (resp. additive) when the right rule in sequent calculus (or introduction rule in natural deduction) for the operator splits the context (resp. copies the whole context in all branches). The example above shows how the additive and multiplicative conjunctions are different. The words multiplicative and additive are originally adjectives but we sometimes use them as nouns: "multiplicatives" for the multiplicative operators and "additives" for the additive operators.

Additive implication can be defined [135, Ch. 4] using additive disjunction, but the multiplicative implication $\multimap$ has a more important role because our interpretation of $\vdash$ symbol in a sequent is the multiplicative implication. This is shown by the following invertible[18] rule:

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \multimap \psi} \quad .$$

In the intuitionistic linear logic, the right side of a sequent can only contain at most one formula. Abramsky [3] gave computational interpretations for the intuitionistic linear logic and the classical linear logic. Since we will extend his linear lambda calculus in Chapter 2, we are going to elaborate on Abramsky's lambda calculus there.

[18]When a rule is changed upside-down and stay admissible, the rule is called *invertible*.

Abelian Logic

Meyer and Slaney [104] and Casari [27] found Abelian logic independently. Meyer and Slaney's motivation was bringing the theory of relevant logics closer to group theory. Casari [27] mentions arguments about comparative propositions in treated in Aristotle's *Topics*:

> x is more (less, as much as) A than y

and so on [27, p. 161]. One semantics of Abelian logic interprets a logical formula as an element of a lattice-ordered Abelian group [57, 3.4.2.]. There, the interpretation of $\mathbf{1}$ is the unit; the interpretation of $\otimes$ is the operation of the group; and the interpretation of $\cdot \multimap \mathbf{1}$ is the inverse element. Abelian logic is an important example in algebraic semantics of logics because Abelian logic is complete with respect to the lattice-ordered group $\mathbb{Z}$ of integers [57, pp. 107-108]. Metcalfe et al. [103] gave a deduction system for Abelian logic, which "almost" [103, after Definition 8] has the subformula property. Metcalfe [102] gave a hypersequent calculus for Abelian logic and proves cut-elimination [102, Theorem 5] for the hypersequent calculus. In Chapter 2, we provide a lambda-calculus based on another hypersequent calculus. Our formulation use conjunctive hypersequents, which allow us to encode a process calculus in our hyper-lambda calculus in a straightforward way (Section 2.4).

Proof Nets and the Amida Lotteries

Proof nets are graphical representation of proofs. Some different sequent calculus proofs are translated into the same proof net because the original proofs are only different in cosmetic ways (like which rule to apply first and exchanges of elements in a sequent). The succinct representation comes with the cost of specifying the correctness condition for such graphical structures. There have been intensive studies on proof nets, which started with Girard [59] and Danos and Regnier [39].

We are going to treat the intuitionistic version of multiplicative linear logic, and its proof nets are based on the Amida lotteries. The Amida lottery is a traditional way to generate permutations on a set in an obfuscated way. Japanese schoolchildren use the *Amida lotteries* for assigning seats, jobs, teams and so on to classmates. The diagram defines a permutation. From a top end, one can find the counterpart by following the vertical line, but one has to cross the bridge whenever one finds one and then after crossing the bridge, one has to go down the vertical line. An example is given in Figure 1.2.

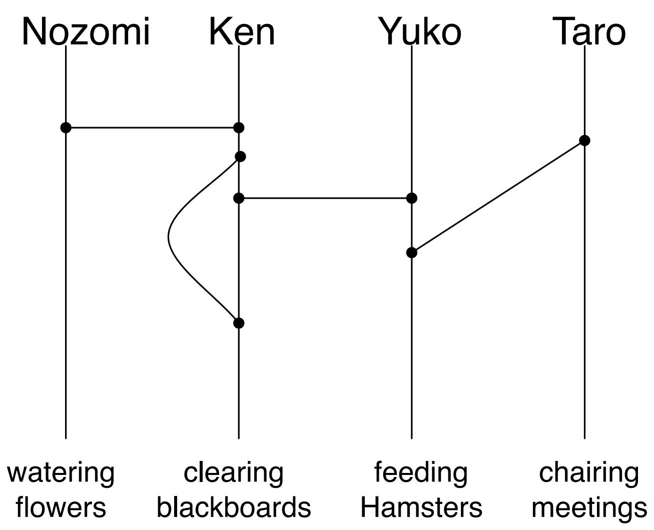


Figure 1.2: An example of Amida lotteries. Nozomi clears blackboards; Ken waters flowers; Yuko chairs meetings; and Taro feeds Hamsters.

1.4 To the Next Chapters

Each chapter can be read independently. Chapter 2 treats a synchronous hyper-lambda calculus and Chapter 3 deals with an asynchronous one. The operational semantics is simpler in the synchronous case while the type system is more exotic in the synchronous case. The asynchronous one is apt for shared-memory implementation. Chapter 4 treats another asynchronous hyper-lambda calculus based on monoidal t-norm logic. In this chapter, we analyze the *prelinearity axiom* $(\varphi \multimap \psi) \oplus (\psi \multimap \varphi)$ using the parametricity argument. Chapter 5 implements a hyper-lambda calculus similar to that in Chapter 3 using Haskell.

Chapter 2

A Synchronous Hyper-Lambda Calculus

2.1 Introduction

There is a PhD student who says:

> I bought a pair of wooden shoes in Amsterdam. I put a coin in the left and a key in the right. Next morning, I found those objects in the opposite shoes: the key in the left and the coin in the right. The same experiments succeeded even if the shoes were put in distance, one in university and one at home. After every night, I found arbitrary objects swapped in the pair of wooden shoes. This fast transfer method might have helped merchants of the Dutch East India Company (VOC) to swap objects between Asia and Europe.

We do not claim existence of such shoes, but propose a similar programming abstraction in the context of typed lambda calculi.

We propose a way to unify ML-style programming languages [98, 108] and π-calculus [107]. "Well-typed expressions do not go wrong," said Milner [106]. However, when communication is involved, how to maintain the principle is not yet settled. For example, Haskell, which is an ML-style programming language, allows different threads to communicate using a kind of shared data store called an MVar `mv` of type `MVar a`, with commands `putMVar mv` of type `a -> IO ()` and `takeMVar mv` of type `IO a`[1]. The former command consumes an argument of type `a` and the consumed argument appears from the latter command. However, if programmers make mistakes, these commands can cause a deadlock during execution even after the program passes type

[1]The arrow `->` shows implication $\supset$ and the tuple () shows the unit type $\mathbf{1}$. In Haskell, a command of type `type a -> IO b` takes an input of type `a` and produces a result of type `b` during execution.

checking. This is because the type system of Haskell allows programmers to use only one of the sender and the receiver. Fundamentally, this is because the type system of Haskell is based on intuitionistic logic, which allows throwing away proofs.

As a remedy, we invent a typed lambda calculus where the user is forced to use both sending and receiving primitives. For that we use the technique of linear types. Linear types are refinements of intuitionistic types. Differently from intuitionistic types, linear types can specify a portion of program to be used just once.

Linear types are used by Wadler [146] and Caires and Pfenning [26] to encode session types, but our type system can type processes that Wadler and Pfenning's system cannot.

As intuitionistic types are based on intuitionistic propositional logic, linear types are based on linear logic. From the intuitionistic linear logic, the only addition is the Amida axiom $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$. We will see that the resulting logic is identical to Abelian logic [27] up to provability of formulae. In the Amida calculus, we can express π-calculus-like processes as macros. Our initial motivation was just studying the axiom $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$[2]. From the viewpoint of typed lambda calculi, a natural way to add the axiom $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$ is to add a pair of primitives c and $\bar{c}$ so that $\cdots ct \cdots \bar{c}u \cdots$ reduces to $\cdots u \cdots t \cdots$: in words, c returns $\bar{c}$'s argument and vice versa. We can obtain the send-receive communication when we specialize the axiom as $(\varphi \multimap \mathbf{1}) \otimes (\mathbf{1} \multimap \varphi)$; the left hand side c of type $\varphi \multimap \mathbf{1}$ is the sending primitive and the right hand side $\bar{c}$ of type $\mathbf{1} \multimap \varphi$ is the receiving primitive. The sending primitive consumes a data of type φ and produces a meaningless[3] data of unit type $\mathbf{1}$. The receiving primitive takes the meaningless data of type $\mathbf{1}$ and produces a data of type φ.

Later in this chapter, we address some questions.

- Due to addition of the Amida axiom, is it the case that every type is inhabited[4]? In other words, is the resulting type system inconsistent? Our answer is no (Theorem 2.5.2).

- Can we generalize the channels to serve more complicated protocols than one-shot send-receive communication? Our answer is yes (Section 2.4).

[2]Takeuti Izumi asked about conjunctions after the author talked about $(\varphi \multimap \psi) \oplus (\psi \multimap \varphi)$.

[3]As we will see, since $\mathbf{1}$ is a provable formula in Abelian logic, the Amida calculus is equipped with a way to obtain a data of type $\mathbf{1}$ for free.

[4]A type φ is inhabited iff there is a closed term t with $\vdash t{:}\varphi$ derivable using the derivation rules in Figure 2.1.

- Can we implement process calculi and session types using these communication primitives? Our answer is yes (Section 2.4).

After following these practical questions, we proceed to developing a proof net structure for the multiplicative fragment of the resulting logic (Section 2.6). Our solution involves the structure of "the Amida lottery," which is a traditional Japanese way of making arbitrary permutations.

2.2 Definitions

2.2.1 Types

We assume a countably infinite set of *propositional variables*, for which we use letters X, Y and so on. We define a type φ by BNF:

$$\varphi ::= \mathbf{1} \mid X \mid \varphi \otimes \varphi \mid \varphi \multimap \varphi \mid \varphi \oplus \varphi \mid \varphi \mathbin{\&} \varphi \ .$$

A *formula* is a type. As the typing rules (Figure 2.1) reveal, $\otimes$ is the multiplicative conjunction, $\multimap$ is the multiplicative implication, $\oplus$ is the additive disjunction and & is the additive conjunction.

2.2.2 Terms and Free Variables

We assume countably infinitely many variables $x, y, z, \ldots$. Before defining terms, following Abramsky's linear lambda calculus LF [3], we define *patterns* binding sets of variables:

- $*$ is a pattern binding $\emptyset$,
- $\langle x, \_\rangle$ and $\langle \_, x\rangle$ are patterns binding $\{x\}$,
- $x \otimes y$ is a pattern binding $\{x, y\}$.

All patterns are from Abramsky's LF [3]. Using patterns, we inductively define a *term* t with *free variables* S. We assume countably infinitely many *channels* with involution satisfying $\bar{c} \neq c$ and $\bar{\bar{c}} = c$.

- $*$ is a term with free variables $\emptyset$,
- a variable x is a term with free variables $\{x\}$,
- if t is a term with free variables S, u is a term with free variables S', and moreover S and S' are disjoint, then $t \otimes u$ and tu are terms with free variables $S \cup S'$,

- if t and u are terms with free variables S, then $\langle t, u\rangle$ is a term with free variables S,
- if t is a term with free variables S, then $\mathsf{inl}(t)$ and $\mathsf{inr}(t)$ are terms with free variables S,
- if t is a term with free variables $S \cup \{x\}$ and x is not in S, then $\lambda x.t$ is a term with free variables S,
- if t is a term with free variables S, p is a pattern binding S', u is a term with free variables $S' \cup S''$ and equalities $S \cap S'' = S' \cap S'' = \emptyset$ hold, then, $\mathsf{let}\, t\, \mathsf{be}\, p\, \mathsf{in}\, u$ is a term with free variables $S \cup S''$,
- if t is a term with free variables S, u is a term with free variables $S'' \cup \{x\}$, v is a term with free variables $S'' \cup \{y\}$, $x, y \notin S''$ and $S \cap S'' = \emptyset$ hold, then

  $$\mathsf{match}\, t\, \mathsf{of}\, \mathsf{inl}(x).u/\mathsf{inr}(y).v$$

  is a term with free variables $S \cup S''$, and
- if t is a term with free variables S, then ct is also a term with free variables S for any channel c.

Only the last clause is original, introducing channels, which are our communication primitives. Note that a term with free variables S is not a term with free variables S' when S and S' are different (even if S is a subset of S'). In other words, the set of free variables $FV(t)$ is uniquely defined for a term t. We introduce an abbreviation

$$\mathsf{ign}\, \epsilon\, \mathsf{in}\, t \equiv t$$

$$\mathsf{ign}\, s_0, \overrightarrow{s}\, \mathsf{in}\, t \equiv \mathsf{let}\, s_0\, \mathsf{be} * \mathsf{in}\, (\mathsf{ign}\, \overrightarrow{s}\, \mathsf{in}\, t)$$

inductively for a sequence of terms $\overrightarrow{s}$. Here ϵ stands for the empty sequence. The symbol ign is intended to be pronounced "ignore."

2.2.3 Typing Derivations

On top of Abramsky's linear lambda calculus LF [3], we add a rule to make a closed term of type $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$. A *context* Γ is a possibly empty sequence of variables associated with types where the same variable appears at most once. A context $x{:}X, y{:}Y$ is allowed, but $x{:}X, x{:}Y$ or $x{:}X, x{:}X$ is not a context. A *hypersequent* is inductively defined as

$$\mathcal{O} ::= \epsilon \mid (\Gamma \vdash t{:}\varphi \;\rule{2pt}{1em}\; \mathcal{O})$$

where Γ is a context. Each $\Gamma \vdash t\!:\!\varphi$ is called a *component* of a hypersequent. In this chapter, we interpret the components conjunctively. Differently from the previous papers [8–10, 12], here, the hypersequent $\Gamma \vdash \varphi \mid \Delta \vdash \psi$ is interpreted as the conjunction of components: $(\bigotimes\Gamma \multimap \varphi) \otimes (\bigotimes\Delta \multimap \psi)$ where $\bigotimes\Gamma$ stands for the $\otimes$-conjunction of elements of Γ. The conjunctive treatment is our original invention, and finding an application of such a treatment is one of our contributions[5]. We name this technique the *conjunctive hypersequent*.

The typing rules of the *Amida calculus* are in Figure 2.1. When $\vdash t\!:\!\varphi$ is derivable, the type φ is *inhabited*.

Example 2.2.1 (Derivation of the Amida axiom) *The type $(\varphi \multimap \psi)\otimes(\psi \multimap \varphi)$ is inhabited by the following derivation.*

$$
{\scriptstyle \otimes R}\dfrac{{\scriptstyle \multimap R}\dfrac{{\scriptstyle \multimap R}\dfrac{{\scriptstyle Sync}\dfrac{{\scriptstyle Merge}\dfrac{{\scriptstyle Ax}\dfrac{}{x\!:\!\varphi \vdash x\!:\!\varphi} \quad {\scriptstyle Ax}\dfrac{}{y\!:\!\psi \vdash y\!:\!\psi}}{x\!:\!\varphi \vdash x\!:\!\varphi \mid y\!:\!\psi \vdash y\!:\!\psi}}{x\!:\!\varphi \vdash cx\!:\!\psi \mid y\!:\!\psi \vdash \bar{c}y\!:\!\varphi}}{\vdash \lambda x.cx\!:\!\varphi \multimap \psi \mid y\!:\!\psi \vdash \bar{c}y\!:\!\varphi}}{\vdash \lambda x.cx\!:\!\varphi \multimap \psi \mid \vdash \lambda y.\bar{c}y\!:\!\varphi \multimap \varphi}}{\vdash (\lambda x.cx)\otimes(\lambda y.\bar{c}y)\!:\!(\varphi \multimap \psi)\otimes(\varphi \multimap \varphi)}
$$

Another example shows how we can type the term $\bar{c}(cx)$.

$$
{\scriptstyle Cut}\dfrac{{\scriptstyle Sync}\dfrac{{\scriptstyle Merge}\dfrac{{\scriptstyle Ax}\dfrac{}{x\!:\!\varphi \vdash x\!:\!\varphi} \quad {\scriptstyle Ax}\dfrac{}{y\!:\!\psi \vdash y\!:\!\psi}}{x\!:\!\varphi \vdash x\!:\!\varphi \mid y\!:\!\psi \vdash y\!:\!\psi}}{x\!:\!\varphi \vdash cx\!:\!\psi \mid y\!:\!\psi \vdash \bar{c}y\!:\!\varphi}}{x\!:\!\varphi \vdash \bar{c}(cx)\!:\!\varphi}
$$

2.2.4 Interaction with Contraction and Weakening

The Amida calculus has two strong properties about the missing structural rules: weakening and contraction. We first state the property about specialized contraction rules. Informally, if one can duplicate a proof of φ into two, then, one can obtain a proof of φ from nothing by first borrowing a proof of φ, second duplicating the proof into two and finally returning one of the two proofs.

Proposition 2.2.2 *For any formula φ, if the specialized contraction rule (here, we omit variables and terms)*

$$
\mathrm{C}\varphi\,\dfrac{\mathcal{O} \mid \varphi,\varphi,\Gamma \vdash \psi}{\mathcal{O} \mid \varphi,\Gamma \vdash \psi}
$$

[5]We have to note however, for Abelian logic, there is an ordinary disjunctive hypersequent system that enjoys cut-elimination. The conjunctive hypersequents reflect some computational intuition.

$$\text{Ax}\;\frac{}{x:\varphi \vdash x:\varphi} \qquad \text{Merge}\;\frac{\mathcal{O} \qquad \mathcal{O}'}{\mathcal{O} \mid \mathcal{O}'} \qquad \text{Cut}\;\frac{\mathcal{O} \mid \Gamma \vdash t:\varphi \mid x:\varphi, \Delta \vdash u:\psi}{\mathcal{O} \mid \Gamma, \Delta \vdash u[t/x]:\psi}$$

$$\text{IE}\;\frac{\mathcal{O} \mid \Gamma, x:\varphi, y:\psi, \Delta \vdash t:\theta}{\mathcal{O} \mid \Gamma, y:\psi, x:\varphi, \Delta \vdash t:\theta} \qquad \text{EE}\;\frac{\mathcal{O} \mid \Gamma \vdash t:\varphi \mid \Delta \vdash u:\psi \mid \mathcal{O}'}{\mathcal{O} \mid \Delta \vdash u:\psi \mid \Gamma \vdash t:\varphi \mid \mathcal{O}'}$$

$$\mathbf{1}\text{R}\;\frac{}{\vdash *:\mathbf{1}} \qquad \mathbf{1}\text{L}\;\frac{\mathcal{O} \mid \Gamma \vdash t:\varphi}{\mathcal{O} \mid \Gamma, z:\mathbf{1} \vdash \mathsf{ign}\,z\,\mathsf{in}\,t:\varphi} \qquad \otimes\text{R}\;\frac{\mathcal{O} \mid \Gamma \vdash t:\varphi \mid \Delta \vdash u:\psi}{\mathcal{O} \mid \Gamma, \Delta \vdash t \otimes u:\varphi \otimes \psi}$$

$$\text{Sync}\;\frac{\mathcal{O} \mid \Gamma \vdash t:\varphi \mid \Delta \vdash u:\psi}{\mathcal{O} \mid \Gamma \vdash ct:\psi \mid \Delta \vdash \bar{c}u:\varphi}\;(c \text{ and } \bar{c} \text{ uniquely introduced here})$$

$$\otimes\text{L}\;\frac{\mathcal{O} \mid \Gamma, x:\varphi, y:\psi \vdash t:\theta}{\mathcal{O} \mid \Gamma, z:\varphi \otimes \psi \vdash \mathsf{let}\,z\,\mathsf{be}\,x \otimes y\,\mathsf{in}\,t:\theta}$$

$$\multimap\text{R}\;\frac{\mathcal{O} \mid \Gamma, x:\varphi \vdash t:\psi}{\mathcal{O} \mid \Gamma \vdash \lambda x.t:\varphi \multimap \psi} \qquad \multimap\text{L}\;\frac{\mathcal{O} \mid \Gamma \vdash t:\varphi \mid x:\psi, \Delta \vdash u:\theta}{\mathcal{O} \mid \Gamma, f:\varphi \multimap \psi, \Delta \vdash u[(ft)/x]:\theta}$$

$$\&\text{R}\;\frac{\Gamma \vdash t:\varphi \qquad \Gamma \vdash u:\psi}{\Gamma \vdash \langle t, u\rangle:\varphi \,\&\, \psi}$$

$$\&\text{L}_0\;\frac{\mathcal{O} \mid \Gamma, x:\varphi \vdash t:\theta}{\mathcal{O} \mid \Gamma, z:\varphi \,\&\, \psi \vdash \mathsf{let}\,z\,\mathsf{be}\,\langle x, \_\rangle\,\mathsf{in}\,t:\theta}$$

$$\&\text{L}_1\;\frac{\mathcal{O} \mid \Gamma, y:\psi \vdash t:\theta}{\mathcal{O} \mid \Gamma, z:\varphi \,\&\, \psi \vdash \mathsf{let}\,z\,\mathsf{be}\,\langle \_, y\rangle\,\mathsf{in}\,t:\theta}$$

$$\oplus\text{R}_0\;\frac{\mathcal{O} \mid \Gamma \vdash t:\varphi}{\mathcal{O} \mid \Gamma \vdash \mathsf{inl}(t):\varphi \oplus \psi} \qquad \oplus\text{R}_1\;\frac{\mathcal{O} \mid \Gamma \vdash u:\psi}{\mathcal{O} \mid \Gamma \vdash \mathsf{inr}(u):\varphi \oplus \psi}$$

$$\oplus\text{L}\;\frac{\Gamma, x:\varphi \vdash u:\theta \qquad \Gamma, y:\psi \vdash v:\theta}{\Gamma, z:\varphi \oplus \psi \vdash \mathsf{match}\,z\,\mathsf{of}\,\mathsf{inl}(x).u/\mathsf{inr}(y).v:\theta}$$

Figure 2.1: The typing rules of the Amida calculus. $\mathcal{O}$ and $\mathcal{O}'$ stand for hypersequents. Most rules are straightforward modification of Abramsky's rules [3]. The Sync rule is original. Rules &R and $\oplus$L are only applicable to singleton hypersequents.

is admissible for any hypersequent $\mathcal{O}$, context Γ and formula ψ, then, φ is inhabited in the Amida calculus.

Proof By the following derivation.

$$\begin{prooftree}
\AxiomC{}
\LeftLabel{$\mathbf{1}$R}
\UnaryInfC{$\vdash *:\mathbf{1}$}
\AxiomC{}
\LeftLabel{Ax}
\UnaryInfC{$x:\varphi \vdash x:\varphi$}
\LeftLabel{Merge}
\BinaryInfC{$\vdash *:\mathbf{1} \mid x:\varphi \vdash x:\varphi$}
\LeftLabel{Sync}
\UnaryInfC{$\vdash c*:\varphi \mid x:\varphi \vdash \bar{c}x:\mathbf{1}$}
\AxiomC{}
\LeftLabel{Ax}
\UnaryInfC{$y:\varphi \vdash y:\varphi$}
\LeftLabel{Merge}
\BinaryInfC{$\vdash c*:\varphi \mid x:\varphi \vdash \bar{c}x:\mathbf{1} \mid y:\varphi \vdash y:\varphi$}
\LeftLabel{$\otimes$R}
\UnaryInfC{$\vdash c*:\varphi \mid x:\varphi, y:\varphi \vdash \bar{c}*\otimes y:\mathbf{1}\otimes\varphi$}
\LeftLabel{Cφ}
\UnaryInfC{$\vdash c*:\varphi \mid x:\varphi \vdash \bar{c}x\otimes x:\mathbf{1}\otimes\varphi$}
\LeftLabel{Cut}
\UnaryInfC{$\vdash (\bar{c}(c*))\otimes(c*):\mathbf{1}\otimes\varphi$}
\AxiomC{}
\LeftLabel{Ax}
\UnaryInfC{$z:\varphi \vdash z:\varphi$}
\LeftLabel{$\mathbf{1}$L}
\UnaryInfC{$k:\mathbf{1}, z:\varphi \vdash \mathsf{ign}\, k \,\mathsf{in}\, z:\varphi$}
\LeftLabel{$\otimes$L}
\UnaryInfC{$z:\mathbf{1}\otimes\varphi \vdash \mathsf{let}\, z \,\mathsf{be}\, k\otimes z \,\mathsf{in}\,\mathsf{ign}\, k \,\mathsf{in}\, z:\varphi$}
\LeftLabel{Merge}
\BinaryInfC{$\vdash (\bar{c}(c*))\otimes(c*):\mathbf{1}\otimes\varphi \mid z:\mathbf{1}\otimes\varphi \vdash \mathsf{let}\, z \,\mathsf{be}\, k\otimes z \,\mathsf{in}\,\mathsf{ign}\, k \,\mathsf{in}\, z:\varphi$}
\LeftLabel{Cut}
\UnaryInfC{$\vdash \mathsf{let}\,(\bar{c}(c*))\otimes(c*)\,\mathsf{be}\, k\otimes z \,\mathsf{in}\,\mathsf{ign}\, k \,\mathsf{in}\, z:\varphi$}
\end{prooftree}$$

where Cφ marks the step where the specialized contraction is used. ■

The admissibility of contraction rule Cφ is equivalent to derivability of $\varphi \vdash \varphi \otimes \varphi$, thanks to $\otimes$R rule and $\otimes$L rule.

Remark 2.2.3 (Incompatibility of the bang modality) *By this proposition, if we are to introduce the bang modality [59] of linear logic, then any formula of the form $!\varphi$ is a theorem. This is because $!\varphi \multimap !\varphi \otimes !\varphi$ is a theorem in intuitionistic linear logic.*

We state another fact, which is known to Casari [27] who wrote "in the ℓ-logic there are no 'additive extrema.'"

Corollary 2.2.4 (Incompatibility of additive disjunctive unit) *If there is a formula $\mathbf{0}$ such that $\mathbf{0} \vdash \varphi$ is derivable for any formula φ, then $\mathbf{0}$ is provable. As a consequence, any formula is provable i.e. the resulting logic is inconsistent.*

Further, this implies that the Amida calculus is incompatible with the second order universal quantification. The second order universal quantification adds a form of type $\forall X \varphi$ and two type derivation rules:

$$\forall\text{R}\ \frac{\mathcal{O} \mid \Gamma \vdash t:\varphi}{\mathcal{O} \mid \Gamma \vdash t:\forall X.\varphi}\ (X \text{ not free in } \mathcal{O} \text{ or } \Gamma)$$

$$\forall\text{L}\ \frac{\mathcal{O} \mid x:\psi[\theta/X], \Gamma \vdash t:\varphi}{\mathcal{O} \mid x:\forall X.\psi, \Gamma \vdash t:\varphi}\quad .$$

Corollary 2.2.5 (Incompatibility of the second order universal quantification) *If we add the second order universal quantification to the Amida calculus, the resulting logic is inconsistent.*

Proof Corollary 2.2.4 is applicable because $x : \forall X.X \vdash x : \varphi$ is derivable for any φ. ∎

Next we state the incompatibility of the weakening rule and Abelian logic. After seeing that the Amida calculus characterizes Abelian logic, the fact is an immediate consequence of completeness of Abelian logic with respect to $\mathbb{Z}$ [104, p. 272], but we can now give a computational interpretation: of the communicating pair $c \colon \varphi \multimap \psi$ and $\bar{c} \colon \psi \multimap \varphi$, with the help of weakening, we can throw away one primitive $\bar{c}$ and still use the other primitive c.

Proposition 2.2.6 (Incompatibility of weakening with Abelian logic) *If we add the weakening rule*

$$\text{W}\,\frac{\mathcal{H} \mid \Gamma \vdash \psi}{\mathcal{H} \mid \varphi, \Gamma \vdash \psi}$$

to Abelian logic (where we omit variables and terms), the resulting logic is inconsistent.

Proof Any formula φ is provable by the following derivation tree.

$$\text{Cut}\,\dfrac{\text{W}\,\dfrac{\multimap\text{R}\,\dfrac{\text{Sync}\,\dfrac{\text{Merge}\,\dfrac{\text{Ax}\,\dfrac{}{\vdash * : \mathbf{1}} \qquad \text{Ax}\,\dfrac{}{x : \varphi \vdash x : \varphi}}{\vdash * : \mathbf{1} \mid x : \varphi \vdash x : \varphi}}{\vdash c* : \varphi \mid x : \varphi \vdash \bar{c}x : \mathbf{1}}}{\vdash c* : \varphi \mid \vdash \lambda x.\bar{c}x : \varphi \multimap \mathbf{1}}}{f : \varphi \multimap \mathbf{1} \vdash c* : \varphi \mid \vdash \lambda x.\bar{c}x : \varphi \multimap \mathbf{1}}}{\vdash c* : \varphi}$$

where W marks the usage of weakening. The resulting term $c*$ sends $*$ somewhere and waits for a response that is never sent back. ∎

To summarize, anything below is inconsistent with the Amida calculus:

- a formula $\mathbf{0}$ such that $\vdash \mathbf{0} \multimap \varphi$ is derivable for all φ,
- the second order universal quantification,
- contraction rule, or
- weakening rule.

Although various extensions to the Amida calculus yield inconsistency, we will see that the Amida calculus itself is consistent because it characterizes Abelian logic (Theorem 2.5.1, Theorem 2.5.2).

2.2.5 Evaluation

As a programming language, the Amida calculus is equipped with an operational semantics that evaluates some closed hyper-terms into a sequence of canonical forms. The *canonical forms* are the same as those of Abramsky's LF [3]:

$$\langle t, u \rangle \qquad * \qquad v \otimes w \qquad \lambda x.t \qquad \mathsf{inl}(v) \qquad \mathsf{inr}(w)$$

where v and w are canonical forms and t and u are terms.

An *evaluation sequence* $\mathcal{E}$ is defined by the following grammar:

$$\mathcal{E} ::= \epsilon \mid (t \Downarrow v \;\,\rule{1pt}{8pt}\;\, \mathcal{E})$$

where v is a canonical form.

Now we define evaluation as a set of evaluation sequences (Figure 2.2). Though most rules are similar to those of Abramsky's LF [3], we add the semantics for channels.

2.3 Type Safety

When we can evaluate a derivable hypersequent, the result is also derivable. Especially, this shows that, whenever a communicating term is used, the communicating term is used according to the types shown in the Sync rule occurrence introducing the communicating term.

Theorem 2.3.1 (Type Preservation of the Amida calculus) *If terms $t_0, \ldots, t_n$ have a hypersequent $\vdash t_0 : \varphi_0 \;\rule{1pt}{8pt}\; \cdots \;\rule{1pt}{8pt}\; \vdash t_n : \varphi_n$ and an evaluation sequence $t_0 \Downarrow v_0 \;\rule{1pt}{8pt}\; \cdots \;\rule{1pt}{8pt}\; t_n \Downarrow v_n$ derivable, then*

$$\vdash v_0 : \varphi_0 \;\rule{1pt}{8pt}\; \cdots \;\rule{1pt}{8pt}\; \vdash v_n : \varphi_n$$

is also derivable.

Proof By induction on evaluation using the propositions below. We classify the situation by the last rule used.

(Merge) By Proposition 2.3.2, we can use the induction hypothesis.

($\mathsf{let}\, t\, \mathsf{be}\, \langle x, \_ \rangle\, \mathsf{in}\, u$) By Proposition 2.3.3, we can use the induction hypothesis.

(Other cases) Similar to above. ■

Two hypersequents $\mathcal{O}$ and $\mathcal{O}'$ are *channel-disjoint* iff it is not the case that $\mathcal{O}$ contains c and $\mathcal{O}'$ contains $\bar{c}$ for any channel c.

$$\frac{}{* \Downarrow *} \qquad \frac{\mathcal{E} \mid t \Downarrow * \mid u \Downarrow v}{\mathcal{E} \mid \mathsf{ign}\, t\, \mathsf{in}\, u \Downarrow v} \qquad \frac{\mathcal{E} \mid t \Downarrow v \mid u \Downarrow w}{\mathcal{E} \mid t \otimes u \Downarrow v \otimes w}$$

$$\frac{\mathcal{E} \mid t \Downarrow v \otimes w \mid u[v/x, w/y] \Downarrow v'}{\mathcal{E} \mid \mathsf{let}\, t\, \mathsf{be}\, x \otimes y\, \mathsf{in}\, u \Downarrow v'}$$

$$\text{Merge}\ \frac{\mathcal{E} \qquad \mathcal{E}'}{\mathcal{E} \mid \mathcal{E}'}$$

(For any channel c, it is not the case that $\mathcal{E}$ contains c and $\mathcal{E}'$ contains $\bar{c}$.)

$$\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{\mathcal{E} \mid t \Downarrow \lambda x.t' \mid u \Downarrow v \mid t'[v/x] \Downarrow w}{\mathcal{E} \mid tu \Downarrow w}$$

$$\frac{\mathcal{E} \mid t \Downarrow v \mid u \Downarrow w}{\mathcal{E} \mid ct \Downarrow w \mid \bar{c}u \Downarrow v}\ (\mathcal{E},\ t \text{ and } u \text{ do not contain } c \text{ or } \bar{c}.)$$

$$\frac{\mathcal{E} \mid t \Downarrow t' \mid s \Downarrow s' \mid \mathcal{E}'}{\mathcal{E} \mid s \Downarrow s' \mid t \Downarrow t' \mid \mathcal{E}'} \qquad \frac{}{\langle t, u\rangle \Downarrow \langle t, u\rangle}$$

$$\frac{\mathcal{E} \mid t \Downarrow \langle t_0, t_1\rangle \mid u[t_0/x] \Downarrow w}{\mathcal{E} \mid \mathsf{let}\, t\, \mathsf{be}\, \langle x, \_\rangle\, \mathsf{in}\, u \Downarrow w}$$

$$\frac{\mathcal{E} \mid t \Downarrow \langle t_0, t_1\rangle \mid u[t_1/y] \Downarrow w}{\mathcal{E} \mid \mathsf{let}\, t\, \mathsf{be}\, \langle \_, y\rangle\, \mathsf{in}\, u \Downarrow w}$$

$$\frac{\mathcal{E} \mid t \Downarrow v}{\mathcal{E} \mid \mathsf{inl}(t) \Downarrow \mathsf{inl}(v)} \qquad \frac{\mathcal{E} \mid u \Downarrow w}{\mathcal{E} \mid \mathsf{inr}(u) \Downarrow \mathsf{inr}(w)}$$

$$\frac{\mathcal{E} \mid t \Downarrow \mathsf{inl}(v) \mid u[v/x] \Downarrow w}{\mathcal{E} \mid \mathsf{match}\, t\, \mathsf{of}\, \mathsf{inl}(x).u/\mathsf{inr}(y).u' \Downarrow w} \qquad \frac{\mathcal{E} \mid t \Downarrow \mathsf{inr}(v) \mid u'[v/y] \Downarrow w}{\mathcal{E} \mid \mathsf{match}\, t\, \mathsf{of}\, \mathsf{inl}(x).u/\mathsf{inr}(y).u' \Downarrow w}$$

Figure 2.2: The definition of evaluation relation of the Amida calculus. $\mathcal{E}$ is possibly the empty evaluation sequence. The whole system is based on Abramsky's LF. Note that the results of evaluation are always canonical forms.

Proposition 2.3.2 (Split) *If a type derivation leading to $\mathcal{O} \mid \mathcal{O}'$ exists for two channel-disjoint hypersequents, both $\mathcal{O}$ and $\mathcal{O}'$ are derivable separately.*

Proof By induction on the type derivation. Notice that all typing rules except Sync touches only one component. Notice also that all typing rules preserve channels from top-to-bottom, i.e., if a component in the assumption of a rule contains a channel, then there is a unique corresponding component in the conclusion of the rule containing the same channel. ■

Proposition 2.3.3 (Inversion on &L) *If* $\mathcal{O} \mathrel{\vert} \Gamma \vdash \mathsf{let}\, t\, \mathsf{be}\, \langle x, \_\rangle\, \mathsf{in}\, u : \theta$ *is derivable, then there is a partition of* Γ *into* Γ_0 *and* Γ_1 *(up to exchange) such that* $\mathcal{O} \mathrel{\vert} \Gamma_0 \vdash t : \varphi \,\&\, \psi \mathrel{\vert} \Gamma_1, x : \varphi \vdash u : \theta$ *is derivable.*

Proof By induction on the original derivation. ■

2.3.1 Lack of Convergence

Convergence states that whenever a closed term t is typed $\vdash t : \varphi$, then an evaluation $t \Downarrow v$ is also derivable for some canonical form v. It is a desirable property so that Abramsky [3] proves it for LF, but there are at least two kinds of counter examples for convergence of the Amida calculus.

Nested Channel Pairs

Consider a typed term:

$$\text{Cut}\,\dfrac{\text{Sync}\,\dfrac{\text{Merge}\,\dfrac{\text{Ax}\,\dfrac{}{x:\mathbf{1}\vdash x:\mathbf{1}} \qquad \oplus\text{R}\,\dfrac{\mathbf{1}\text{R}\,\dfrac{}{\vdash *:\mathbf{1}}}{\vdash \mathsf{inl}(*):\mathbf{1}\oplus\mathbf{1}}}{x:\mathbf{1}\vdash x:\mathbf{1} \mathrel{\vert} \vdash \mathsf{inl}(*):\mathbf{1}\oplus\mathbf{1}}}{x:\mathbf{1}\vdash cx:\mathbf{1}\oplus\mathbf{1} \mathrel{\vert} \vdash \bar{c}(\mathsf{inl}(*)):\mathbf{1}}}{\vdash c(\bar{c}(\mathsf{inl}(*))):\mathbf{1}\oplus\mathbf{1}}$$

with no evaluation. When we think about why this process does not have an evaluation, one explanation is this process is deadlocked. In order to evaluate this closed term, we can add the following eval-subst rule:

$$\text{eval-subst}\,\dfrac{\mathcal{E} \mathrel{\vert} t \Downarrow v \mathrel{\vert} u[v/x] \Downarrow w}{\mathcal{E} \mathrel{\vert} u[t/x] \Downarrow w}$$

so that the following evaluation is possible

$$\text{eval-subst}\,\dfrac{\dfrac{\dfrac{\dfrac{}{* \Downarrow *} \qquad \dfrac{\dfrac{}{* \Downarrow *}}{\mathsf{inl}(*) \Downarrow \mathsf{inl}(*)}}{* \Downarrow * \mathrel{\vert} \mathsf{inl}(*) \Downarrow \mathsf{inl}(*)}}{c* \Downarrow \mathsf{inl}(*) \mathrel{\vert} \bar{c}(\mathsf{inl}(*)) \Downarrow *}}{c(\bar{c}(\mathsf{inl}(*))) \Downarrow \mathsf{inl}(*)} \;.$$

However, adding the eval-subst rule breaks the current proof of Theorem 2.3.1 (safety), but with some modifications, the safety property can possibly be proved. Even if we try that, we must face the real difficulty for finding an operational semantics with convergence.

Channels Sending Bound Variables

Consider a typed term:

$$\multimap\text{R}\ \cfrac{\text{Sync}\ \cfrac{\text{Merge}\ \cfrac{\text{Ax}\ \cfrac{}{x\!:\!X \vdash x\!:\!X} \qquad \mathbf{1}\text{R}\ \cfrac{}{\vdash *\!:\!\mathbf{1}}}{x\!:\!X \vdash x\!:\!X \mid \vdash *\!:\!\mathbf{1}}}{x\!:\!X \vdash cx\!:\!\mathbf{1} \mid \vdash \bar{c}*\!:\!X}}{\vdash \lambda x.cx\!:\!X \multimap \mathbf{1} \mid \vdash \bar{c}*\!:\!X} \quad .$$

If convergence holds, there must be an evaluation

$$\lambda x.cx \Downarrow \lambda x.cx \mid \bar{c}* \Downarrow v \quad .$$

When the right component $\bar{c}* \Downarrow v$ is introduced, the introduction must use an assumption of the form $x \Downarrow v$. However, open terms have no evaluation. Explicit substitutions [1] might be helpful here. Note that, although the above example poses a difficulty for convergence, the above example does not witness a deadlock.

Regaining Convergence

Since any thunk[6] can be evaluated, we can enclose any typed term t of type φ within a thunk $\lambda x.t$ of type $\mathbf{1} \multimap \varphi$ so that $\lambda x.t \Downarrow \lambda x.t$ is derivable.

2.3.2 Determinacy

Determinacy states that if $t \Downarrow v$ and $t \Downarrow w$ hold, then v and w are identical. Since our evaluation is given to possibly multiple terms at the same time, it is easier to prove a more general version.

Theorem 2.3.4 (General Determinacy of the Amida calculus) *If*

$$t_0 \Downarrow v_0 \mid t_1 \Downarrow v_1 \mid \cdots \mid t_n \Downarrow v_n$$

and

$$t_0 \Downarrow w_0 \mid t_1 \Downarrow w_1 \mid \cdots \mid t_n \Downarrow w_n$$

hold, then each v_i is identical to w_i.

Proof By induction on the height of evaluation derivation. Each component in the conclusion has only one applicable rule. Also, the order of decomposing different components is irrelevant (the crucial condition is freshness of c and $\bar{c}$ in Figure 2.2).■

[6]A thunk is a lambda abstraction $\lambda x.t$ whose type is $\mathbf{1} \multimap \varphi$.

2.4 Session Types and Processes as Abbreviations

In order to see the usefulness of the communication primitives, we try implementing a process calculus and a session type system using the Amida calculus.

2.4.1 Session Types as Abbreviations

As an abbreviation, we introduce *session types*. Session types [78, 131] can specify a communication protocol over a channel. The following definitions and the descriptions are modification from Wadler's translations and descriptions of session types [146]. The notation here is different from the original notation by Takeuchi, Honda and Kubo [131].

$!\varphi\,\psi \equiv \varphi \multimap \psi$	output a value of φ then behave as ψ
$?\varphi\,\psi \equiv \varphi \otimes \psi$	input a value of φ then behave as ψ
$\oplus\{l_i\colon \varphi_i\}_{i\in I} \equiv \varphi_0 \,\&\, \cdots \,\&\, \varphi_n, \quad I = \{0,\ldots,n\}$	select from φ_i with label l_i
$\&\{l_i\colon \varphi_i\}_{i\in I} \equiv \varphi_0 \oplus \cdots \oplus \varphi_n, \quad I = \{0,\ldots,n\}$	offer choice of φ_i with label l_i
$\mathsf{end} \equiv \mathbf{1}$	terminator

where I is a finite downward-closed set of natural numbers like $\{0, 1, 2, 3\}$. As Wadler [146] notes, the encoding looks opposite of what some would expect, but as Wadler [146] explains, we are typing channels instead of processes.

The grammar

$$\varphi, \psi ::= \mathsf{end} \mid X \mid !\varphi\,\psi \mid ?\varphi\,\psi \mid \oplus\{l_i\colon \varphi_i\}_{i\in I} \mid \&\{l_i\colon \varphi_i\}_{i\in I}$$

covers all types.

A linear type ($\varphi^{\sim}$ possibly with subscript) is generated by this grammar:

$$\varphi^{\sim} ::= \mathsf{end} \mid !\psi\,\varphi^{\sim} \mid ?\psi\,\varphi^{\sim} \mid \oplus\{l_i\colon \varphi_i^{\sim}\}_{i\in I} \mid \&\{l_i\colon \varphi_i^{\sim}\}_{i\in I}$$

We define duals of linear types. Again the definition is almost the same as Wadler [146]'s except that end is self-dual.

$$\begin{aligned}
\overline{!\psi\,\varphi^{\sim}} &= ?\psi\,\overline{\varphi^{\sim}} \\
\overline{?\psi\,\varphi^{\sim}} &= !\psi\,\overline{\varphi^{\sim}} \\
\overline{\oplus\{l_i\colon \varphi_i^{\sim}\}_{i\in I}} &= \&\{l_i\colon \overline{\varphi_i^{\sim}}\}_{i\in I} \\
\overline{\&\{l_i\colon \varphi_i^{\sim}\}_{i\in I}} &= \oplus\{l_i\colon \overline{\varphi_i^{\sim}}\}_{i\in I} \\
\overline{\mathsf{end}} &= \mathsf{end} \ .
\end{aligned}$$

2.4.2 Processes as Abbreviations

We define the sending and receiving constructs of process calculi as abbreviations:

$x\langle u\rangle.\, t \equiv t[(xu)/x]$	send u through channel x and then use x in t
$x(y).\, t \equiv \mathsf{let}\, x\, \mathsf{be}\, y \otimes x\, \mathsf{in}\, t$	receive y through channel x and use x and y in t
$0 \equiv *$	do nothing

We have to be careful about substitution combined with process abbreviations. For example, $(x\langle u\rangle.\, t)[s/x]$ is not $s\langle u\rangle.\, t$ because the latter is not defined. Following the definition, $(x\langle u\rangle.\, t)[s/x]$ is actually $(t[xu/x])[s/x] = t[su/x]$. We are going to introduce the name restriction $\nu x.t$ after implementing channels.

Below, we are going to justify these abbreviations statically and dynamically. The static justification comes from typing rules (Figure 2.4.3) and the dynamic justification comes from evaluation (Figure 2.4.4).

2.4.3 Process Typing Rules as Abbreviations

The session type abbreviation and the processes abbreviation allow us to use the typing rules in the next proposition.

Theorem 2.4.1 (Process Typing Rules: senders and receivers) *These rules are admissible.*

$$\text{recv}\ \frac{\mathcal{O} \mathrel{\Vert} y\!:\!\psi, x\!:\!\chi \vdash t\!:\!\varphi}{\mathcal{O} \mathrel{\Vert} x\!:\,?\psi\,\chi \vdash x(y).\, t\!:\!\varphi} \qquad \text{send}\ \frac{\mathcal{O} \mathrel{\Vert} \Gamma, x\!:\!\chi \vdash t\!:\!\varphi \mathrel{\Vert} \Delta \vdash u\!:\!\psi}{\mathcal{O} \mathrel{\Vert} \Gamma, \Delta, x\!:\,!\psi\,\chi \vdash x\langle u\rangle.\, t\!:\!\varphi}$$

$$\text{end}\ \frac{\mathcal{O} \mathrel{\Vert} \Gamma \vdash t\!:\!\varphi}{\mathcal{O} \mathrel{\Vert} \Gamma, x\!:\!\mathsf{end} \vdash \mathsf{ign}\, x\, \mathsf{in}\, t\!:\!\varphi} \qquad \frac{}{\vdash 0\!:\!\mathbf{1}}$$

Before presenting the proof, we note that the types of variable x change in the rules. This reflects the intuition of session types: the session type of a channel changes after some communication occurs through the channel.

Proof After expanding abbreviations, the first rule is actually one of the original rules:

$$\otimes\text{L}\ \frac{\mathcal{O} \mathrel{\Vert} y\!:\!\psi, x\!:\!\chi \vdash t\!:\!\varphi}{\mathcal{O} \mathrel{\Vert} x\!:\!\psi \otimes \chi \vdash \mathsf{let}\, x\, \mathsf{be}\, y \otimes x\, \mathsf{in}\, t\!:\!\varphi}\ .$$

After expanding abbreviations, the second rule is also one of the original rules:

$$\multimap\text{L}\ \frac{\mathcal{O} \mathrel{\Vert} \Gamma, x\!:\!\chi \vdash t\!:\!\varphi \mathrel{\Vert} \Delta \vdash u\!:\!\psi}{\mathcal{O} \mathrel{\Vert} \Gamma, \Delta, x\!:\!\psi \multimap \chi \vdash t[(xu)/x]\!:\!\varphi}\ .$$

The third and the fourth rule are more evidently one of the original rules **1**L and **1**R.■

Example 2.4.2 (Typed communicating terms) *Using Theorem 2.4.1, we can type processes. Figure 2.3 contains one process, which sends a channel y through x and then waits for input in a channel y'. Here is another process that takes an input w' from channel x', where the input w' itself is expected to be a channel. After receiving w', the process puts* $\mathsf{inl}(*)$ *in w'.*

$$
\mathit{recv}\,\dfrac{\mathit{end}\,\dfrac{\mathit{send}\,\dfrac{\mathit{end}\,\dfrac{\mathbf{1}R\,\dfrac{}{\vdash *:\mathbf{1}}}{w':\mathsf{end}\vdash \mathsf{ign}\,w'\,\mathsf{in}\,*:\mathbf{1}} \qquad \oplus R\,\dfrac{\mathbf{1}R\,\dfrac{}{\vdash *:\mathbf{1}}}{\vdash \mathsf{inl}(*):\mathbf{1}\oplus\mathbf{1}}}{w':!(\mathbf{1}\oplus\mathbf{1})\,\mathsf{end}\vdash w'\langle\mathsf{inl}(*)\rangle.\,\mathsf{ign}\,w'\,\mathsf{in}\,*:\mathbf{1}}}{w':!(\mathbf{1}\oplus\mathbf{1})\,\mathsf{end}, x':\mathsf{end}\vdash \mathsf{ign}\,x'\,\mathsf{in}\,w'\langle\mathsf{inl}(*)\rangle.\,\mathsf{ign}\,w'\,\mathsf{in}\,*:\mathbf{1}}}{x':?(!(\mathbf{1}\oplus\mathbf{1})\,\mathsf{end})\,\mathsf{end}\vdash x'(w').\,\mathsf{ign}\,x'\,\mathsf{in}\,w'\langle\mathsf{inl}(*)\rangle.\,\mathsf{ign}\,w'\,\mathsf{in}\,*:\mathbf{1}}
$$

We intend these two processes to communicate when we connect the channels x and x'. For that, we have to implement complicated channels as in the following subsection.

2.4.4 Implementing Channels

We introduced primitives c and $\bar{c}$ implementing $(\mathbf{1}\multimap\varphi)\otimes(\varphi\multimap\mathbf{1})$. These can be seen as channels of session types $?\varphi\,\mathsf{end}$ and $!\varphi\,\mathsf{end}$. Indeed, $?\varphi\,\mathsf{end}$ is $\varphi\otimes\mathbf{1}$ (which is inter-derivable with $\mathbf{1}\multimap\varphi$) and $!\varphi\,\mathsf{end}$ is $\varphi\multimap\mathbf{1}$. We can generalize this phenomenon to the more complicated session types[7].

Proposition 2.4.3 (Session realizers) *For any linear type $\varphi^{\sim}$, the hypersequent*

$$\vdash t:\varphi^{\sim} \mid \vdash u:\overline{\varphi^{\sim}}$$

is derivable for some terms t and u.

Proof Induction on $\varphi^{\sim}$.

(end) $\text{Merge}\,\dfrac{\text{Ax}\,\dfrac{}{\vdash *:\mathbf{1}} \qquad \text{Ax}\,\dfrac{}{\vdash *:\mathbf{1}}}{\vdash *:\mathbf{1} \mid \vdash *:\mathbf{1}}$ is what we seek.

($!\psi\,\varphi^{\sim}$) By the induction hypothesis, $\vdash t':\varphi^{\sim} \mid \vdash u':\overline{\varphi^{\sim}}$ is derivable. Using this, we can make the following derivation:

$$
\dfrac{\otimes\text{R}\,\dfrac{\text{Sync}\,\dfrac{\text{Merge}\,\dfrac{\text{Ax}\,\dfrac{}{x:\psi\vdash x:\psi} \qquad \overset{\text{IH}}{\vdash t':\varphi^{\sim} \mid \vdash u':\overline{\varphi^{\sim}}}}{x:\psi\vdash x:\psi \mid \vdash t':\varphi^{\sim} \mid \vdash u':\overline{\varphi^{\sim}}}}{cx:\varphi^{\sim}\vdash x:\psi \mid \vdash \bar{c}t':\psi \mid \vdash u':\overline{\varphi^{\sim}}}}{x:\psi\vdash cx:\varphi^{\sim} \mid \vdash (\bar{c}t')\otimes u':\psi\otimes\overline{\varphi^{\sim}}}}{\vdash \lambda x.cx:\psi\multimap\varphi^{\sim} \mid \vdash (\bar{c}t')\otimes u':\psi\otimes\overline{\varphi^{\sim}}}\ .
$$

[7]This is impossible using the ordinary linear types.

$$
\text{send}\;\dfrac{
\text{recv}\;\dfrac{
\text{end}\;\dfrac{
\text{end}\;\dfrac{
\text{Ax}\;\dfrac{}{z : \mathbf{1} \oplus \mathbf{1} \vdash z : \mathbf{1} \oplus \mathbf{1}}
}{z : \mathbf{1} \oplus \mathbf{1}, y : \mathsf{end} \vdash \mathsf{ign}\, y \,\mathsf{in}\, z : \mathbf{1} \oplus \mathbf{1}}
}{z : \mathbf{1} \oplus \mathbf{1}, x : \mathsf{end}, y : \mathsf{end} \vdash \mathsf{ign}\, x, y \,\mathsf{in}\, z : \mathbf{1} \oplus \mathbf{1}}
}{x : \mathsf{end}, y : ?(\mathbf{1} \oplus \mathbf{1})\,\mathsf{end} \vdash y(z).\,\mathsf{ign}\, x, y \,\mathsf{in}\, z : \mathbf{1} \oplus \mathbf{1}}
\qquad
\text{Ax}\;\dfrac{}{y' : !(\mathbf{1} \oplus \mathbf{1})\,\mathsf{end} \vdash y' : !(\mathbf{1} \oplus \mathbf{1})\,\mathsf{end}}
}{y : ?(\mathbf{1} \oplus \mathbf{1})\,\mathsf{end}, x : !(!(\mathbf{1} \oplus \mathbf{1})\,\mathsf{end})\,\mathsf{end}, y' : !(\mathbf{1} \oplus \mathbf{1})\,\mathsf{end} \vdash x\langle y\rangle.\, y'(z).\,\mathsf{ign}\, x, y \,\mathsf{in}\, z : \mathbf{1} \oplus \mathbf{1}}
$$

Figure 2.3: A typed process.

($?\psi\,\varphi$) Symmetric to the above.

($\oplus\{l_i\colon \varphi_i\}$) By the induction hypothesis, for each $i \in I$, we have

$$\vdash t_i : \varphi_i \;\Big\Vert\; \vdash u_i : \overline{\varphi_i}$$

derived. Hence derivable is

$$\vdash t_i : \varphi_i \;\Big\Vert\; \vdash i(u_i) : \oplus_{j\in I}\overline{\varphi_j}$$

where $i(u_i)$ is an appropriate nesting of $\mathsf{inl}(\cdot)$, $\mathsf{inr}(\cdot)$ and u_i. Combining $|I|$ such derivations, we can derive

$$\vdash \langle t_i\rangle_{i\in I} : \&_{i\in I}\varphi_i \;\Big\Vert\; \langle \vdash i(u_i) : \oplus_{j\in I}\overline{\varphi_j}\rangle_{i\in I}$$

for a fresh natural number n.

($\&\{l_i\colon \varphi_i\}$) Symmetric to above. ■

We call the above pair t, u in the statement the *session realizers* of $\varphi^\sim$ and denote them by $\rhd(\varphi^\sim), \lhd(\varphi^\sim)$. Moreover, we use $\rhd\!\lhd(\varphi^\sim)$ to denote the pair $\rhd(\varphi^\sim) \otimes \lhd(\varphi^\sim)$. So far, a free variable with a linear type represented a channel serving the corresponding session type. Now, we can substitute the free variables with the session realizers so that the typed processes can actually communicate. If we have two terms that use free variables of type $\varphi^\sim$ and $\overline{\varphi^\sim}$, we can replace those free variables by session realizers.

Corollary 2.4.4 (Binding both ends of a channel) *If*

$$\mathcal{O} \;\Big\Vert\; \Gamma, x : \varphi^\sim \vdash t : \psi \;\Big\Vert\; \Delta, y : \overline{\varphi^\sim} \vdash u : \theta$$

is derivable,

$$\mathcal{O} \;\Big\Vert\; \Gamma \vdash t[\rhd(\varphi^\sim)/x] : \psi \;\Big\Vert\; \Delta \vdash u[\lhd(\varphi^\sim)/y] : \theta$$

is also derivable.

Now we can define the name restriction operator as an abbreviation:

$$\nu x : \varphi^\sim . t \equiv \mathsf{let}\, \rhd\!\lhd(\varphi^\sim)\, \mathsf{be}\, x_L \otimes x_R\, \mathsf{in}\, t$$

where we assume injections $x \mapsto x_L$ and $x \mapsto x_R$ whose images are disjoint.

Then, in addition to Theorem 2.4.1, more typing rules are available.

Theorem 2.4.5 (Process typing rule: name restriction) *The following typing rule is admissible.*

$$\frac{\mathcal{O} \mathrel{\rule{2pt}{0.9em}} \Gamma, x\colon\varphi^{\sim}, y\colon\overline{\varphi^{\sim}} \vdash t\colon\psi}{\mathcal{O} \mathrel{\rule{2pt}{0.9em}} \Gamma \vdash \nu x\colon\varphi^{\sim}.t[x_L/x][x_R/y]\colon\psi}$$

Example 2.4.6 (Connecting processes using session realizers) *Using the session realizers, we can connect the processes typed in Example 2.4.2. Indeed,*

$$\begin{array}{l}\vdash \nu(x\colon ?(!(\mathbf{1}\oplus\mathbf{1})\,\mathsf{end})\,\mathsf{end}).\nu(y\colon !(\mathbf{1}\oplus\mathbf{1})\,\mathsf{end}).\\ \quad (x_R\langle y_L\rangle.\, y_R(z).\,\mathsf{ign}\, x_R, y_R\,\mathsf{in}\, z) \otimes (x_L(w').\,\mathsf{ign}\, x_L\,\mathsf{in}\, w'\langle\mathsf{inl}(*)\rangle.\,\mathsf{ign}\, w'\,\mathsf{in}\, *)\\ \quad : (\mathbf{1}\oplus\mathbf{1})\otimes\mathbf{1}\end{array}$$

is derivable.

Now we have to check the evaluation of the term in this example. For that we prepare a lemma.

Process Evaluation as Abbreviation

The intention of defining $x\langle u\rangle.\,t_0$ and $y(z).\,t_1$ is mimicking communication in process calculi. When we substitute x and y with session type realizers, these terms can actually communicate.

The next lemma can help us evaluate session realizers.

Lemma 2.4.7 *Let t_0 be a term containing a free variable x and t_1 be a term containing free variables y and z. The rule*

$$\frac{\mathcal{E} \mathrel{\rule{2pt}{0.9em}} \triangleright(\varphi^{\sim}) \Downarrow v' \mathrel{\rule{2pt}{0.9em}} \triangleleft(\varphi^{\sim}) \Downarrow w' \mathrel{\rule{2pt}{0.9em}} t_0[v'/x] \Downarrow v \mathrel{\rule{2pt}{0.9em}} u \Downarrow u' \mathrel{\rule{2pt}{0.9em}} t_1[u'/z][w'/y] \Downarrow w}{\begin{array}{c}\mathcal{E} \mathrel{\rule{2pt}{0.9em}} \triangleright(!\psi\, \varphi^{\sim}) \Downarrow \lambda x.cx \mathrel{\rule{2pt}{0.9em}} \triangleleft(!\psi\, \varphi^{\sim}) \Downarrow u' \otimes w' \mathrel{\rule{2pt}{0.9em}}\\ (x\langle u\rangle.\, t_0)[\lambda x.cx/x] \Downarrow v \mathrel{\rule{2pt}{0.9em}} (y(z).\, t_1)[u'\otimes w'/y] \Downarrow w\end{array}}$$

is admissible under presence of the eval-subst rule.

Proof By the derivation in Figure 2.4.

Example 2.4.8 (Evaluation of communicating processes) *Here is an example of evaluation using the eval-subst rule.*

$$\dfrac{\begin{array}{c}\diamondsuit\;\dfrac{\dfrac{\dfrac{}{\triangleright(\mathsf{end}) \Downarrow * \mathrel{\rule{2pt}{0.9em}} \triangleleft(\mathsf{end}) \Downarrow *} \quad \dfrac{\dfrac{\dfrac{}{* \Downarrow *}}{\mathsf{inl}(*) \Downarrow \mathsf{inl}(*)} \quad \dfrac{}{* \Downarrow *}}{\mathsf{inl}(*) \Downarrow \mathsf{inl}(*) \mathrel{\rule{2pt}{0.9em}} * \Downarrow *} \quad \dfrac{\dfrac{}{* \Downarrow *}}{\mathsf{inl}(*) \Downarrow \mathsf{inl}(*)}}{\triangleright(\mathsf{end}) \Downarrow * \mathrel{\rule{2pt}{0.9em}} \triangleleft(\mathsf{end}) \Downarrow * \mathrel{\rule{2pt}{0.9em}} \mathsf{inl}(*) \Downarrow \mathsf{inl}(*) \mathrel{\rule{2pt}{0.9em}} * \Downarrow * \mathrel{\rule{2pt}{0.9em}} \mathsf{inl}(*) \Downarrow \mathsf{inl}(*)}}{\triangleright(!(\mathbf{1}\oplus\mathbf{1})\,\mathsf{end}) \Downarrow \lambda x.cx \mathrel{\rule{2pt}{0.9em}} \triangleleft(!(\mathbf{1}\oplus\mathbf{1})\,\mathsf{end}) \Downarrow \mathsf{inl}(*) \otimes * \mathrel{\rule{2pt}{0.9em}}}\\ (x_L\langle\mathsf{inl}(*)\rangle.\,\mathsf{ign}\, x_L\,\mathsf{in}\, *)[\lambda x.cx/x_L] \Downarrow * \mathrel{\rule{2pt}{0.9em}} (x_R(z).\,\mathsf{ign}\, x_R\,\mathsf{in}\, z)[\mathsf{inl}(*) \otimes */x_R] \Downarrow \mathsf{inl}(*)\end{array}}{\dfrac{\begin{array}{c}\triangleright\triangleleft(!(\mathbf{1}\oplus\mathbf{1})\,\mathsf{end}) \Downarrow \lambda x.cx \otimes (\mathsf{inl}(*) \otimes *) \mathrel{\rule{2pt}{0.9em}}\\ (x_L\langle\mathsf{inl}(*)\rangle.\,\mathsf{ign}\, x_L\,\mathsf{in}\, *)[\lambda x.cx/x_L] \otimes (x_R(z).\,\mathsf{ign}\, x_R\,\mathsf{in}\, z)[\mathsf{inl}(*) \otimes */x_R] \Downarrow * \otimes \mathsf{inl}(*)\end{array}}{\nu(x\colon !(\mathbf{1}\oplus\mathbf{1})\,\mathsf{end}).\,(x_L\langle\mathsf{inl}(*)\rangle.\,\mathsf{ign}\, x_L\,\mathsf{in}\, *) \otimes (x_R(z).\,\mathsf{ign}\, x_R\,\mathsf{in}\, z) \Downarrow * \otimes \mathsf{inl}(*)}}$$

$$\frac{\mathcal{E} \mid \rhd(\varphi^{\sim}) \Downarrow v' \mid \lhd(\varphi^{\sim}) \Downarrow w' \mid t_0[v'/x] \Downarrow v \mid u \Downarrow u' \mid t_1[u'/z][w'/y] \Downarrow w}{\mathcal{E} \mid \rhd(\varphi^{\sim}) \Downarrow v' \mid \lhd(\varphi^{\sim}) \Downarrow w' \mid t_0[v'/x] \Downarrow v \mid u \Downarrow u' \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}$$

$$\frac{\overline{\lambda x.cx \Downarrow \lambda x.cx} \qquad \mathcal{E} \mid \rhd(\varphi^{\sim}) \Downarrow v' \mid \lhd(\varphi^{\sim}) \Downarrow w' \mid t_0[v'/x] \Downarrow v \mid u \Downarrow u' \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}{\mathcal{E} \mid \rhd(\varphi^{\sim}) \Downarrow v' \mid \lhd(\varphi^{\sim}) \Downarrow w' \mid t_0[v'/x] \Downarrow v \mid \lambda x.cx \Downarrow \lambda x.cx \mid u \Downarrow u' \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}$$

$$\frac{\mathcal{E} \mid \rhd(\varphi^{\sim}) \Downarrow v' \mid \lhd(\varphi^{\sim}) \Downarrow w' \mid t_0[v'/x] \Downarrow v \mid \lambda x.cx \Downarrow \lambda x.cx \mid u \Downarrow u' \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}{\mathcal{E} \mid \bar{c}(\rhd(\varphi^{\sim})) \Downarrow u' \mid \lhd(\varphi^{\sim}) \Downarrow w' \mid t_0[v'/x] \Downarrow v \mid \lambda x.cx \Downarrow \lambda x.cx \mid cu \Downarrow v' \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}$$

$$\frac{\mathcal{E} \mid \bar{c}(\rhd(\varphi^{\sim})) \Downarrow u' \mid \lhd(\varphi^{\sim}) \Downarrow w' \mid t_0[v'/x] \Downarrow v \mid \lambda x.cx \Downarrow \lambda x.cx \mid cu \Downarrow v' \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}{\mathcal{E} \mid \bar{c}(\rhd(\varphi^{\sim})) \Downarrow u' \mid \lhd(\varphi^{\sim}) \Downarrow w' \mid t_0[v'/x] \Downarrow v \mid (\lambda x.cx)u \Downarrow v' \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}$$

$$\text{eval-subst}\ \frac{\mathcal{E} \mid \bar{c}(\rhd(\varphi^{\sim})) \Downarrow u' \mid \lhd(\varphi^{\sim}) \Downarrow w' \mid t_0[v'/x] \Downarrow v \mid (\lambda x.cx)u \Downarrow v' \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}{\mathcal{E} \mid \bar{c}(\rhd(\varphi^{\sim})) \Downarrow u' \mid \lhd(\varphi^{\sim}) \Downarrow w' \mid t_0[(\lambda x.cx)u/x] \Downarrow v \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}$$

$$\frac{\mathcal{E} \mid \bar{c}(\rhd(\varphi^{\sim})) \Downarrow u' \mid \lhd(\varphi^{\sim}) \Downarrow w' \mid t_0[(\lambda x.cx)u/x] \Downarrow v \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}{\mathcal{E} \mid \bar{c}(\rhd(\varphi^{\sim})) \otimes \lhd(\varphi^{\sim}) \Downarrow u' \otimes w' \mid t_0[(\lambda x.cx)u/x] \Downarrow v \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}$$

$$\frac{\overline{\lambda x.cx \Downarrow \lambda x.cx} \qquad \mathcal{E} \mid \bar{c}(\rhd(\varphi^{\sim})) \otimes \lhd(\varphi^{\sim}) \Downarrow u' \otimes w' \mid t_0[(\lambda x.cx)u/x] \Downarrow v \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}{\mathcal{E} \mid \lambda x.cx \Downarrow \lambda x.cx \mid \bar{c}(\rhd(\varphi^{\sim})) \otimes \lhd(\varphi^{\sim}) \Downarrow u' \otimes w' \mid t_0[(\lambda x.cx)u/x] \Downarrow v \mid \mathsf{let}\, u' \otimes w' \,\mathsf{be}\, z \otimes y \,\mathsf{in}\, t_1 \Downarrow w}$$

Figure 2.4: Proof of Lemma 2.4.7. The conclusion is identical to our goal up to abbreviations.

The step $\diamondsuit$ uses Lemma 2.4.7.

2.4.5 Copycatting

Proposition 2.4.9 *For any linear type $\varphi^{\sim}$, we can derive $x : \varphi^{\sim}, y : \overline{\varphi^{\sim}} \vdash t : \mathbf{1}$ for some term t.*

Proof By induction on $\varphi^{\sim}$.

(end) Since $\overline{\mathsf{end}} = \mathsf{end}$, the derivation

$$\dfrac{\dfrac{\dfrac{}{\vdash * : \mathbf{1}}}{y : \mathsf{end} \vdash \mathsf{ign}\, y \,\mathsf{in}\, * : \mathbf{1}}}{x : \mathsf{end}, y : \mathsf{end} \vdash \mathsf{ign}\, x, y \,\mathsf{in}\, * : \mathbf{1}}$$

suffices.

($!\psi\, \varphi^{\sim}$) Using the induction hypothesis (IH.), we obtain a derivation:

$$\dfrac{\dfrac{\dfrac{\vdots\ \text{(IH.)}}{x : \varphi^{\sim}, y : \overline{\varphi^{\sim}} \vdash t : \mathbf{1}} \qquad \dfrac{}{z : \psi \vdash z : \psi}}{x : !\psi\, \varphi^{\sim}, y : \overline{\varphi^{\sim}}, z : \psi \vdash x\langle z\rangle.\, t : \mathbf{1}}}{x : !\psi\, \varphi^{\sim}, y : ?\psi\, \overline{\varphi^{\sim}} \vdash y(z).\, x\langle z\rangle.\, t : \mathbf{1}}$$

($?\psi\, \varphi^{\sim}$) Symmetric to above.

($\oplus\{l_i : \varphi_i^{\sim}\}$) For brevity, we only consider a binary disjunction of the form $\oplus\{\text{Left} : \varphi_0^{\sim}, \text{Right} : \varphi_1^{\sim}\}$. By the induction hypotheses, $x_0 : \varphi_0^{\sim}, y_0 : \overline{\varphi_0^{\sim}} \vdash t_0 : \mathbf{1}$ and $x_1 : \varphi_1^{\sim}, y_1 : \overline{\varphi_1^{\sim}} \vdash t_1 : \mathbf{1}$ are derivable. Thus,

$$x_0 : \varphi_0^{\sim}, y : \overline{\varphi_0^{\sim} \oplus \varphi_1^{\sim}} \vdash \mathsf{let}\, y \,\mathsf{be}\, \langle y_0, \_\rangle \,\mathsf{in}\, t_0 : \mathbf{1}$$

and

$$x_1 : \varphi_1^{\sim}, y : \overline{\varphi_0^{\sim} \oplus \varphi_1^{\sim}} \vdash \mathsf{let}\, y \,\mathsf{be}\, \langle \_, y_1\rangle \,\mathsf{in}\, t_1 : \mathbf{1}$$

are derivable. Using these, we can conclude a derivation of

$$\begin{aligned}&x : \varphi_0^{\sim} \oplus \varphi_1^{\sim}, y : \overline{\varphi_0^{\sim} \oplus \varphi_1^{\sim}} \vdash \\ &\mathsf{match}\, x \,\mathsf{of}\, \mathsf{inl}(x_0).\mathsf{let}\, y \,\mathsf{be}\, \langle y_0, \_\rangle \,\mathsf{in}\, t_0 / \mathsf{inr}(x_1).\mathsf{let}\, y \,\mathsf{be}\, \langle \_, y_1\rangle \,\mathsf{in}\, t_1 : \mathbf{1}\ \ .\end{aligned}$$

($\&\{l_i : \varphi_i^{\sim}\}$) Symmetric to above. ■

2.5 Correctness with Respect to Abelian Logic

In this section, we compare the Amida calculus and Abelian logic and discover the fact that they are identical. This characterization provides a straightforward counter-example for cut-elimination. First, we show that all Abelian logic theorems are inhabited in the Amida calculus (completeness of the Amida calculus). After that, we show that all types inhabited in the Amida calculus are theorems of Abelian logic.

2.5.1 Completeness of the Amida Calculus Type System

According to Metcalfe et al. [103], Abelian logic is axiomatized by the following axioms **A1** to **A10** and deduction rules **mp** and &**I**. An abbreviation $\varphi \multimapboth \psi$ is defined to denote $(\varphi \multimap \psi) \mathbin{\&} (\psi \multimap \varphi)$.

A1 $((\varphi \oplus \psi) \multimap \theta) \multimapboth ((\varphi \multimap \theta) \mathbin{\&} (\psi \multimap \theta))$

A2 $((\varphi \otimes \psi) \multimap \theta) \multimapboth (\varphi \multimap (\psi \multimap \theta))$

A3 $(\varphi \multimap \psi) \multimap ((\psi \multimap \theta) \multimap (\varphi \multimap \theta))$

A4 $((\varphi \multimap \psi) \mathbin{\&} (\varphi \multimap \theta)) \multimap (\varphi \multimap (\psi \mathbin{\&} \theta))$

A5 $(\varphi \mathbin{\&} (\psi \oplus \theta)) \multimap ((\varphi \mathbin{\&} \psi) \oplus (\varphi \mathbin{\&} \theta))$

A6 $\varphi \multimapboth (\mathbf{1} \multimap \varphi)$

A7 $(\varphi \mathbin{\&} \psi) \multimap \varphi$

A8 $(\varphi \mathbin{\&} \psi) \multimap \psi$

A9 $\varphi \multimap ((\varphi \multimap \psi) \multimap \psi)$

A10 $((\varphi \multimap \psi) \multimap \psi) \multimap \varphi$

mp if $\varphi \multimap \psi$ and φ are theorems, ψ is also a theorem

&**I** if φ and ψ are theorems, $\varphi \mathbin{\&} \psi$ is also a theorem.

The most peculiar of these axioms is **A10**, which is called the relativisation axiom.

Theorem 2.5.1 (Completeness of the Amida Calculus for Abelian Logic) *A theorem of Abelian logic is inhabited in the Amida calculus.*

Proof All axioms **A1** to **A10** are inhabited in the Amida calculus and the Amida calculus rules satisfy the properties **mp** and &**I**. In particular, a term with type **A10** can be derived as

$$
\multimap\mathrm{R}\,\dfrac{\multimap\mathrm{L}\,\dfrac{\multimap\mathrm{R}\,\dfrac{\mathrm{Sync}\,\dfrac{\mathrm{Merge}\,\dfrac{\mathrm{Ax}\,\dfrac{}{x:\varphi\vdash x:\varphi}\qquad \mathrm{Ax}\,\dfrac{}{y:\psi\vdash y:\psi}}{x:\varphi\vdash x:\varphi \;\boldsymbol{|}\; y:\psi\vdash y:\psi}}{x:\varphi\vdash cx:\psi \;\boldsymbol{|}\; y:\psi\vdash \bar{c}y:\varphi}}{\vdash \lambda x.cx:\varphi\multimap\psi \;\boldsymbol{|}\; y:\psi\vdash \bar{c}y:\varphi}}{z:(\varphi\multimap\psi)\multimap\psi\vdash \bar{c}(z(\lambda x.cx)):\varphi}}{\vdash \lambda z.\bar{c}(z(\lambda x.cx)):((\varphi\multimap\psi)\multimap\psi)\multimap\varphi}\quad . \qquad \blacksquare
$$

2.5.2 Soundness of the Amida Calculus Type System

For soundness, a different axiomatization of Abelian logic in [102] is more useful.

L1 $\varphi \multimap \varphi$

L2 $(\varphi \multimap \psi) \multimap ((\psi \multimap \theta) \multimap (\varphi \multimap \theta))$

L3 $(\varphi \multimap (\psi \multimap \theta)) \multimap (\psi \multimap (\varphi \multimap \theta))$

L4 $((\varphi \otimes \psi) \multimap \theta) \multimapboth (\varphi \multimap (\psi \multimap \theta))$

L5 $(\varphi \,\&\, \psi) \multimap \varphi$

L6 $(\varphi \,\&\, \psi) \multimap \psi$

L7 $((\varphi \multimap \psi) \,\&\, (\varphi \multimap \theta)) \multimap (\varphi \multimap (\psi \,\&\, \theta))$

L8 $\varphi \multimap (\varphi \oplus \psi)$

L9 $\psi \multimap (\varphi \oplus \psi)$

L10 $((\varphi \multimap \theta) \,\&\, (\psi \multimap \theta)) \multimap ((\varphi \oplus \psi) \multimap \theta)$

L11 $\varphi \multimapboth (\mathbf{1} \multimap \varphi)$

L12 $((\varphi \multimap \mathbf{1}) \multimap \mathbf{1}) \multimap \varphi$

C1 $\mathbf{1} \multimapboth (\mathbf{1} \otimes \mathbf{1})$

C2 $(\varphi \multimap \varphi) \multimap \mathbf{1}$

mp If $\varphi \multimap \psi$ and φ are theorems, ψ is also a theorem

&I If φ and ψ are theorems, $\varphi \,\&\, \psi$ is also a theorem.

These axioms and rules are enough to prove all inhabited types of the Amida calculus.

Theorem 2.5.2 (Soundness of the Amida Calculus for Abelian Logic) *An inhabited type in the Amida calculus is a theorem of Abelian logic.*

Proof The IMALL deduction rules are straightforwardly obtained by combining **L1**–**L12**, &**I** and **mp**. The Amida axiom $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$ is a theorem in Abelian logic by the following argument.

1. By **C2**, $((\varphi \multimap \mathbf{1}) \multimap (\varphi \multimap \mathbf{1})) \multimap \mathbf{1}$ is a theorem of Abelian logic.
2. By **L4**, **L5** and **mp**, $(((\varphi \multimap \mathbf{1}) \otimes \varphi) \multimap \mathbf{1}) \multimap ((\varphi \multimap \mathbf{1}) \multimap (\varphi \multimap \mathbf{1}))$ is a theorem of Abelian logic.
3. By 1., 2., **L2** and **mp**, $(((\varphi \multimap \mathbf{1}) \otimes \varphi) \multimap \mathbf{1}) \multimap \mathbf{1}$ is a theorem of Abelian logic.
4. By 3., **L12** and **mp**, $(\varphi \multimap \mathbf{1}) \otimes \varphi$ is a theorem of Abelian logic.
5. By a symmetric argument, $(\psi \multimap \mathbf{1}) \otimes \psi$ is also a theorem of Abelian logic.
6. By 4., 5., **L1**, **L4** and **mp**, $((\varphi \multimap \mathbf{1}) \otimes \varphi) \otimes ((\psi \multimap \mathbf{1}) \otimes \psi)$ is a theorem of Abelian logic.
7. Since Abelian logic contains IMALL without additive units, $((\varphi \multimap \mathbf{1}) \otimes \psi) \multimap (\varphi \multimap \psi)$ is a theorem of Abelian logic.
8. By a symmetric argument, $((\psi \multimap \mathbf{1}) \otimes \varphi) \multimap (\psi \multimap \varphi)$ is also a theorem of Abelian logic.
9. Combining 6., 7., 8., a theorem $(\varphi \multimap \psi) \multimap (\theta \multimap \tau) \multimap (\varphi \otimes \theta) \multimap (\psi \otimes \tau)$ and mp, we obtain $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$ as a theorem of Abelian logic. ■

As a corollary of soundness and completeness, some previous literature (Casari [27], Meyer and Slaney [104]) on Abelian logic provides some facts.

Corollary 2.5.3 (Division by two) *If $\varphi \otimes \varphi$ is inhabited, so is φ.*

Corollary 2.5.4 (Excluded Middle in the Amida calculus) *The law of excluded middle $\varphi \otimes (\varphi \multimap \mathbf{1})$ is inhabited in the Amida calculus.*

Corollary 2.5.5 (Prelinearity in the Amida calculus) *Prelinearity $(\varphi \multimap \psi) \oplus (\psi \multimap \varphi)$ is inhabited in the Amida calculus.*

2.6 Proof Nets

Toward better understanding the Amida calculus, a technique called proof nets seems promising. Generally, proof nets are straightforward for the multiplicative fragments but complicated when additive and exponential connectives are involved. Since the Amida axiom $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$ does not contain additives $(\&, \oplus)$ or exponentials $(!, ?)$, we can focus on the multiplicative connectives ($\multimap$ and $\otimes$). The fragment is called *IMLL* (intuitionistic multiplicative fragment of linear logic). We also use the unit $\mathbf{1}$ for technical reasons. We first describe the IMLL proof nets and their properties. Then we add a new kind of edges called the Amida edges and see it characterizes Abelian logic.

The Amida links are named after the Amida lottery (see Subsection 1.3.9).

2.6.1 IMLL Essential Nets

Proof nets for intuitionistic linear logics are called *essential nets*. This subsection reviews some known results about the essential nets for intuitionistic multiplicative linear logic (IMLL). The exposition here is strongly influenced by Murawski and Ong [111].

We can translate a polarity $p \in \{+, -\}$ and an IMLL formula φ into a *polarized MLL formula* $\ulcorner\varphi\urcorner^p$ following Lamarche [92] and Murawski and Ong [111]. We omit the definition of polarized MLL formulae because the whole grammar is exposed in the translation below:

$$\begin{aligned}
\ulcorner\mathbf{1}\urcorner^+ &= \mathbf{1}^+ & \ulcorner\mathbf{1}\urcorner^- &= \bot^- \\
\ulcorner X\urcorner^+ &= X^+ & \ulcorner X\urcorner^- &= X^- \\
\ulcorner\varphi \multimap \psi\urcorner^+ &= \ulcorner\varphi\urcorner^- \parr^+ \ulcorner\psi\urcorner^+ & \ulcorner\varphi \multimap \psi\urcorner^- &= \ulcorner\varphi\urcorner^+ \otimes^- \ulcorner\psi\urcorner^- \\
\ulcorner\varphi \otimes \psi\urcorner^+ &= \ulcorner\varphi\urcorner^+ \otimes^+ \ulcorner\psi\urcorner^+ & \ulcorner\varphi \otimes \psi\urcorner^- &= \ulcorner\varphi\urcorner^- \parr^- \ulcorner\psi\urcorner^- \quad .
\end{aligned}$$

For example, the Amida axiom can be translated into a polarized MLL formula

$$\begin{aligned}
&\ulcorner(X \multimap Y) \otimes (Y \multimap X)\urcorner^+ \\
&= \ulcorner X \multimap Y\urcorner^+ \otimes^+ \ulcorner Y \multimap X\urcorner^+ \\
&= \left(\ulcorner X\urcorner^- \parr^+ \ulcorner Y\urcorner^+\right) \otimes^+ \left(\ulcorner Y\urcorner^- \parr^+ \ulcorner X\urcorner^+\right) \\
&= \left(X^- \parr^+ Y^+\right) \otimes^+ \left(Y^- \parr^+ X^+\right) \quad .
\end{aligned}$$

The symbol $\parr$ is pronounced “parr.”

Any polarized MLL formula can be translated further into a finite rooted tree containing these branches and polarized atomic formulae $(X^-, X^+, \mathbf{1}^+, \bot^-)$ at the leaves.

$$\begin{tikzcd}
\varphi^- & & \psi^+ \\
\varphi^- & \parr^+ \arrow[ul, dash, dashed] \arrow[ur] & \psi^+
\end{tikzcd}
\quad
\begin{tikzcd}
\varphi^- \arrow[dr] & & \psi^- \arrow[dl] \\
\varphi^- & \parr^- & \psi^-
\end{tikzcd}
\quad
\begin{tikzcd}
\varphi^+ & & \psi^+ \\
\varphi^+ & \otimes^+ \arrow[ul] \arrow[ur] & \psi^+
\end{tikzcd}
\quad
\begin{tikzcd}
\varphi^+ & & \psi^- \arrow[dl] \\
\varphi^+ & \otimes^- \arrow[ul] & \psi^-
\end{tikzcd}$$

For brevity, we sometimes write only the top connectives of labeling formulae. In that case, these branching nodes above are denoted like this.

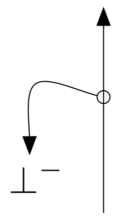
$$\begin{tikzcd}
- & & + \\
& \parr^+ \arrow[ul, dash, dashed] \arrow[ur] &
\end{tikzcd}
\quad
\begin{tikzcd}
- \arrow[dr] & & - \arrow[dl] \\
& \parr^- &
\end{tikzcd}
\quad
\begin{tikzcd}
+ & & + \\
& \otimes^+ \arrow[ul] \arrow[ur] &
\end{tikzcd}
\quad
\begin{tikzcd}
+ & & - \arrow[dl] \\
& \otimes^- \arrow[ul] &
\end{tikzcd}$$

We call arrows with upward (resp. downward) signs *up-edges* (resp. *down-edges*). The *dashed child* of a $\parr^+$ node p is the node which the dashed line from p reaches. The branching nodes labeled by $\parr^+, \parr^-, \otimes^+$ and $\otimes^-$ are called *operator nodes.*

When we add axiom edges and $\perp$-branches (shown below) to the other operator nodes (shown above) we obtain an *essential net* of φ. Due to the arbitrariness of choosing axiom edges and $\perp$-branches, there are possibly multiple essential nets for a formula[8].

$$\begin{tikzcd}
X^+ \arrow[r, bend left] & X^-
\end{tikzcd}$$

axiom edge



$\perp$-branch

Example 2.6.1 (An essential net of the Amida axiom) *Here is one of the essential nets of the Amida axiom* $(X^- \parr^+ Y^+) \otimes^+ (Y^- \parr^+ X^+)$.

$$\begin{tikzcd}
X^- & Y^+ \arrow[r] & Y^- & X^+ \arrow[lll, bend right] \\
& \parr^+ \arrow[ul, dash, dashed] \arrow[u] & & \parr^+ \arrow[ul, dash, dashed] \arrow[u] \\
& & \otimes^+ \arrow[ull] \arrow[ur] &
\end{tikzcd}$$

[8]Murawski and Ong [111] restricts the class of formulae to linearly balanced formulae so that the essential net is uniquely determined.

However, the essential net in Example 2.6.1 is rejected by the following correctness criterion.

Definition 2.6.2 (Correct essential nets) *A correct essential net is an essential net satisfying all these conditions:*

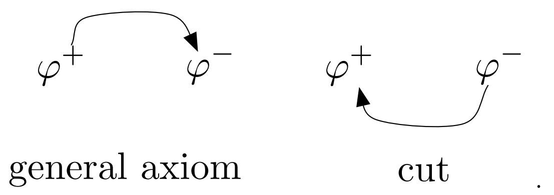
1. *Any node labeled with X^+ (resp. Y^-) is connected to a unique node labeled with X^- (resp. Y^+). Any leaf labeled with $\bot^-$ is connected to a $\bot$-branch. $\mathbf{1}^+$ is not connected to anything above itself;*
2. *the directed graph formed by up-edge, down-edge, axiom edges and $\bot$-branches is acyclic;*
3. *for every $⅋^+$-node p, every path[9] from the root that reaches p's dashed child also passes through p.*

The essential net in Example 2.6.1 is not correct for condition 3. Actually, the Amida axiom does not have any correct essential net. IMLL sequent calculus has the subformula property so that we can confirm that the Amida axiom is not provable in IMLL.

Theorem 2.6.3 (Essential nets by Lamarche [92], Murawski and Ong [111]) *An IMLL formula φ is provable in IMLL iff there exists a correct essential net of φ.*

Proof The left to right is relatively easy. For the other way around, Lamarche [92] uses a common technique of decomposing an essential net from the bottom. Murawski and Ong [111] chose to reduce the problem to sequents of special forms called regular. ∎

Actually, Lamarche [92] also considers the cut rule[10] in essential nets, thus we can include the following general axioms (as macros) and cuts (as primitives) and still use Theorem 2.6.3:

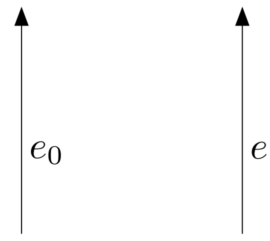


[9]A path must follow solid edges according to the direction. Dashed edges are not directed and they are not contained in paths.
[10]As well as additive operators and exponentials.

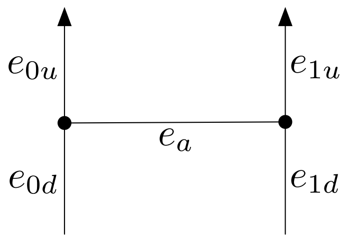
2.6.2 The Amida Nets

Definition 2.6.4 (The Amida nets) *For a hypersequent $\mathcal{H}$, Amida nets of $\mathcal{H}$ are inductively defined by the following three clauses:*

- *an essential net of $\mathcal{H}$ is an Amida net of $\mathcal{H}$;*
- *for an Amida net of $\mathcal{H}$ with two different*[11] *up-edges,*

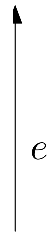


  replacing these with



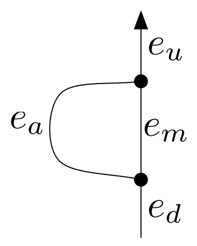
  yields an Amida net of $\mathcal{H}$, where the above component has two paths $e_{0d}e_ae_{1u}$ and $e_{1d}e_ae_{0u}$;

- *for an Amida net of $\mathcal{H}$ with an up-edge,*



  replacing this with

[11] The two edges can be connected by a new edge as long as they are different; their relative positions do not matter.

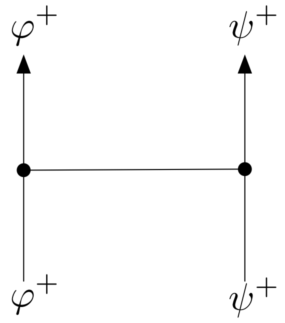


yields an Amida net of $\mathcal{H}$*, where the above component has one finite path* $e_d e_a e_u$ *and one infinite path* $\cdots e_m e_a e_m e_a \cdots$.

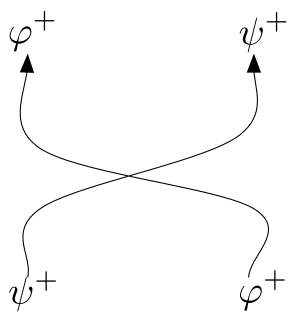
In these clauses, we call the edges labeled e_a the *Amida edges*.

Definition 2.6.5 (Correct Amida nets) *A correct Amida net is an Amida net satisfying the three conditions in Definition 2.6.4.*

The Amida edge is not merely a crossing of up-edges. See the difference between



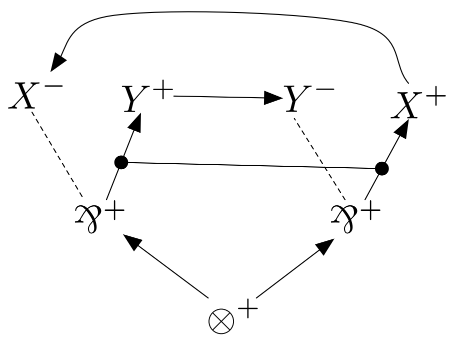
and



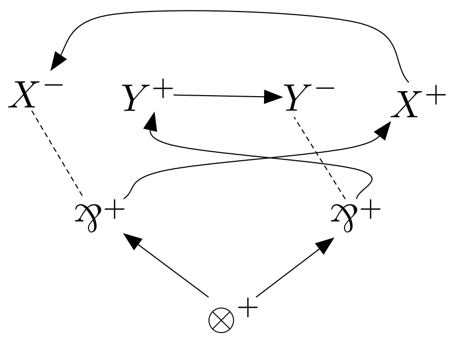
.

The difference is the labels at the bottom. Although Amida edges cross the paths, they do not transfer labels. This difference of labels makes Amida nets validate the Amida axiom.

Example 2.6.6 (A correct Amida net for the Amida axiom) *Here is a correct Amida net for the Amida axiom* $(X \multimap Y) \otimes (Y \multimap X)$.



In terms of the set of paths, the above Amida net is equivalent to the following correct essential net for $(X \multimap X) \otimes (Y \multimap Y)$.

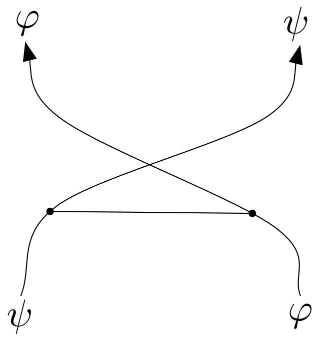


2.6.3 Soundness and Completeness of Amida nets

Theorem 2.6.7 (Completeness of Amida nets) *If a hypersequent* $\mathcal{H}$ *is derivable, there is a correct Amida net for* $\mathcal{H}$.

Proof Inductively on hypersequent derivations. The Sync rule is translated into a crossing with an Amida edge:

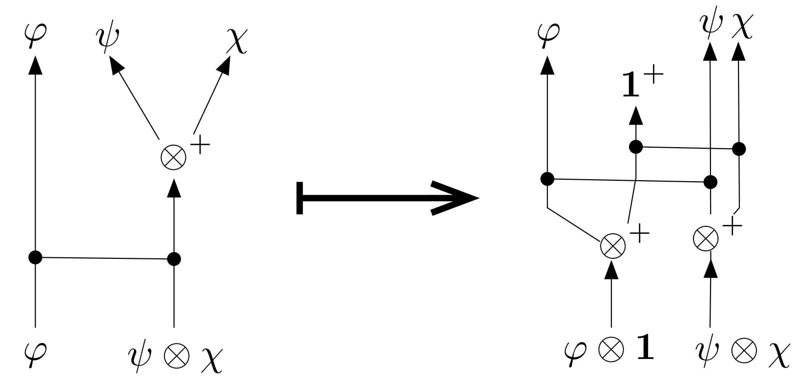
$$\frac{\Gamma \vdash \varphi \;\big|\; \Delta \vdash \psi}{\Gamma \vdash \psi \;\big|\; \Delta \vdash \varphi} \quad \mapsto$$

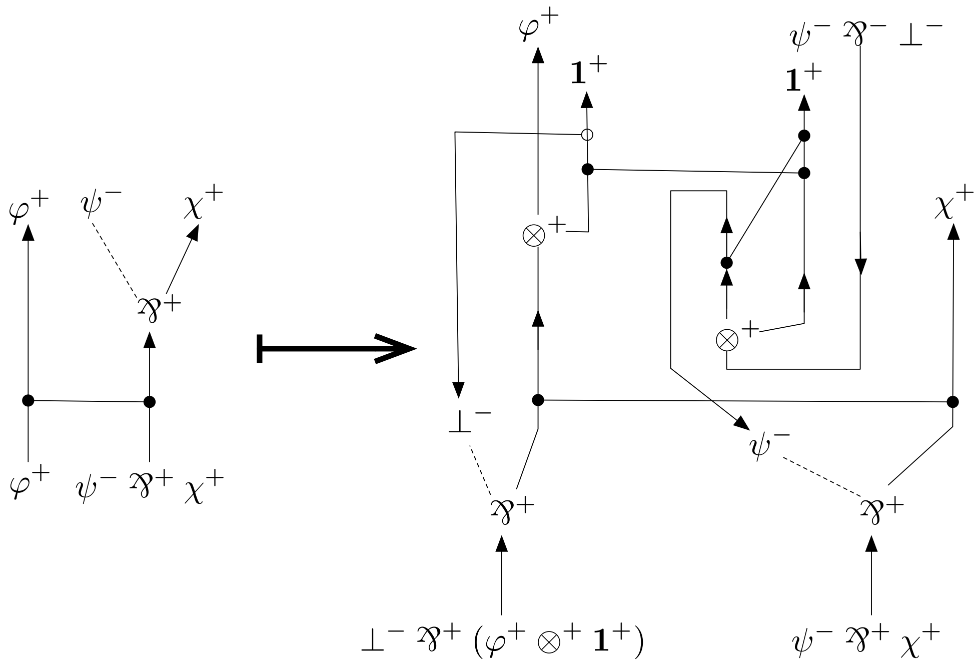


where the crossing exchanges the formulae and the Amida edge keeps the path connections vertically straight. ■

Theorem 2.6.8 (Soundness of Amida nets) *If there is a correct Amida net for a hypersequent $\mathcal{H}$, then $\mathcal{H}$ is derivable.*

Proof From a correct Amida net, first we move the Amida edges upwards until they are just below axiom edges[12]. The moves are as follows.





These translations have two properties.

[12]The idea is similar to the most popular syntactic cut-elimination proofs.

1. When the original (contained in a larger picture) is a correct Amida net, the translation (contained in the same larger picture) is also a correct Amida net.
2. The end nodes of the translation correspond to the end nodes of the original, and the corresponding end nodes have the same label (up to logical equivalence in IMLL). In case of $\otimes$ translation, φ and $\varphi \otimes \mathbf{1}$ are logically equivalent because $\mathbf{1}$ is the unit of $\otimes$. In case of $⅋$ translation, φ and $\bot ⅋ \varphi$ are logically equivalent because $\bot$ is the unit of $⅋$.

For checking the first condition, it is enough to follow the paths (crossing all Amida edges). For the second condition, it is enough to follow the vertical edges ignoring the Amida edges and $\bot$-branches.

The $\otimes$ move introduces Amida edges only above the branching rules. Although the $⅋$ move introduces an Amida edge below a branching rule, that branching rule is of $\otimes$ nature. Also, the $⅋$ move introduces an Amida edge below ψ-axiom link, which is actually a macro. So we have to continue applying the translation moves in the macro. However, since ψ is a strictly smaller subformula of $\psi ⅋ \chi$, this does not cause infinite recursion.

Then, by these translation moves, the whole Amida net is decomposed vertically into three layers. At the top, there is a layer with only axiom edges. In the middle, there is a layer with only vertical edges and Amida edges. At the bottom, there is a layer that contains only ordinary essential net nodes.

Since the middle layer is an Amida lottery, it defines a permutation. That permutation can be expressed as a product of transpositions, so that the original Amida lottery is equivalent to an encoding of a hypersequent derivation that consists of only Sync rules.

After we encode the top and the middle layer into a hypersequent derivation, encoding the bottom layer can be done in the same way as Lamarche's approach [92]. ■

We wonder whether it is possible to add Amida edges to the IMALL$^-$ essential nets following Lamarche [92]. The additives are notoriously difficult for proof nets and we do not expect the combination of additive connectives and Amida edges can be treated in a straightforward way.

2.7 Related Work

2.7.1 Analytic Calculus for Abelian Logic

Metcalfe et al. [103] gave a labeled sequent calculus for Abelian logic. Some years later, Metcalfe [102] gave a hypersequent calculus for Abelian logic and proved cut-elimination theorem for the hypersequent calculus. His formulation is different from ours because Metcalfe's system does not use conjunctive hypersequents. He sticks to the traditional hypersequent formulation where components are interpreted disjunctively. The paper [102] is informative about some logics weaker than Abelian logic so that there might be a hint of getting a hyper-lambda calculi for these weaker logics.

Before Metcalfe et al. [103], Shirahata [128] studied the multiplicative fragment of Abelian logic, which he called CMLL (compact multiplicative linear logic). He gave a categorical semantics for the proofs of a sequent calculus presentation of CMLL and then proved that the cut-elimination procedure of the sequent calculus preserves the semantics. Since our Amida net presentation also gives a non-degenerate semantics for the same logic, we speculate that Amida nets provide a suitable presentation for morphisms of compact closed categories. Remarkably, Shirahata [128] also noted that addition of infinite additives yields inconsistency although he did not notice his CMLL is identical to the multiplicative fragment of Abelian logic[13]. He also noted that addition of finite additives yields a counter-example of cut-elimination, which we will note in Subsection 2.8.2.

2.7.2 Linear Logic and Session Types

Session types [78, 131] appeared in this chapter. Session types aim at typing channels and processes in π-calculus so that the process execution is deterministic and not ending in deadlock. In general, session type systems are not based on a well-known logic. However, there are recent developments on encoding session types into linear type systems.

Kobayashi-Pierce-Turner's Type System

Kobayashi et al. [90] developed a type system for the π-calculus processes. Similarly to the type system presented here, their type system can specify types of communication contents through a name and how many times a name can be used. In some sense, that type system is more flexible than the one shown in this chapter. First, their type system allows a liberal typing on a channel so that the channel can be used for any number of times. Second, their type system can type replicated processes. Third, their type system allows weakening [90, Lemma 3.2]. In other respects, the type system in

[13]Ciabattoni et al. [31] already pointed out the fact that Shirahata [128] and Metcalfe et al. [103] studied the same logic.

[90] is less expressible. That type system does not have lambda abstractions. Also, in contrast to our type system, it is impossible to substitute a free variable with a process in that type system. This is related to the fact that in their type system, a sequent contains types on the left side but not on the right side. Since there are no types on the right side, their sequent is hard to interpret logically.

Caires and Pfenning's Typing System

Caires and Pfenning [26] provide a type system for a fragment of π-calculus. Their type system imposes a discipline stronger than necessary to provide deadlock freedom. For example, this escrowing process P below is not typable in their type system:

$$P = x\langle y\rangle.\, x(a).\, y(b).\, x\langle b\rangle.\, y\langle a\rangle.\, 0 \ \ .$$

The process first emits a channel y through channel x and then takes inputs from x and y and outputs them respectively to y and x. Following the informal description of types by Caires and Pfenning [26], the process P should be typable as

$$\vdash P :: x : (A \multimap B) \otimes (B \multimap A) \ \ .$$

However, such typing is not possible because $(A \multimap B) \otimes (B \multimap A)$ is not a theorem of dual intuitionistic linear logic (DILL), which their type system is based on. In our type system, the following sequent is derivable

$$\begin{aligned}&x : !(!B\ ?A\ \mathsf{end})\ ?B\ !A\ \mathsf{end} \vdash \\ &\nu(y : !B\ ?A\ \mathsf{end}).x\langle y_L\rangle.\, x(a).\, y_R(b).\, x\langle b\rangle.\, y_R\langle a\rangle.\, \mathsf{ign}\ x, y\ \mathsf{in}\ 0 : \mathbf{1}\end{aligned}$$

The resulting sequent indicates that the process is typable with one open channel x that first emits a channel that one can send B to and receive A from, second receives a value of B and finally sends a value of A. This example shows that our type system can type some terms that the type system in Caires and Pfenning [26] cannot.

The most complicated example in Caires and Pfenning [26] involves a drink server.

Example 2.7.1 (Drink server from Caires and Pfenning [26] in the Amida cal.)

$$\begin{aligned}\mathit{ServerProto} &= (N \multimap I \multimap (N \otimes \mathbf{1}))\ \&\ (N \multimap (I \otimes \mathbf{1})) \\ &= (!N\ !I\ ?N\ \mathsf{end})\ \&\ (!N\ ?I\ \mathbf{1})\end{aligned}$$

N stands for the type of strings and I stands for the type of integers, but following Caires and Pfenning [26], we identify both N and I with $\mathbf{1}$. Below, SP abbreviates

ServerProto. Here is the process of the server, which serves one client and terminates.

$$Serv = \langle s(pn).\, s(cn).\, s\langle rc\rangle.\, \mathsf{ign}\, pn, cn, s\, \mathsf{in}\, 0, \quad s(pn).\, s\langle pr\rangle.\, \mathsf{ign}\, s, pn\, \mathsf{in}\, 0\rangle$$

We can derive a sequent $s : \overline{SP} \vdash Serv : \mathbf{1}$.

Here is one client:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash 0 : \mathbf{1}}}{s : \mathsf{end} \vdash \mathsf{ign}\, s\, \mathsf{in}\, 0 : \mathbf{1}}}{s : \mathsf{end}, pr : I \vdash \mathsf{ign}\, pr, s\, \mathsf{in}\, 0 : \mathbf{1}}}{s : ?I\ \mathsf{end} \vdash s(pr).\, \mathsf{ign}\, pr, s\, \mathsf{in}\, 0 : \mathbf{1}} \qquad \dfrac{}{\vdash tea : N}}{\dfrac{s : !N\ ?I\ \mathsf{end} \vdash s\langle tea\rangle.\, s(pr).\, \mathsf{ign}\, pr, s\, \mathsf{in}\, 0 : \mathbf{1}}{s : ServerProto \vdash \mathsf{let}\, s\, \mathsf{be}\, \langle \_, s\rangle\, \mathsf{in}\, s\langle tea\rangle.\, s(pr).\, \mathsf{ign}\, pr, s\, \mathsf{in}\, 0 : \mathbf{1}}}$$

In words, the client first chooses the server's second protocol, which is price quoting, and asks the price of the tea, receives the price and terminates.

We can combine the server with this client. However, since the Amida calculus lacks the exponential modality, Amida calculus cannot type any term with $!ServerProto$*, which the type system of Caires and Pfenning can [26]. In order to do that, we might want to tolerate inconsistency and add* μ *and* ν *operators from the modal* μ*-calculus, like Baelde [14] did, and express* $!ServerProto$ *as* $\nu X.(SeverProto \otimes X)$.

Wadler's Encoding

Wadler [146] gave a type system for a process calculus based on classical linear logic. Although the setting is classical, the idea is more or less the same as Caires and Pfenning [26]. Wadler's type system cannot type the escrowing process above. Worse, the definition of processes by Wadler [146] does not recognize the escrowing process

$$x\langle y\rangle.\, x(a).\, y(b).\, x\langle b\rangle.\, y\langle a\rangle.\, 0$$

as a process at all. His grammar requires the output construction to be used in the form

$$x\langle y\rangle.\, (P \mid Q)$$

where x is bound in Q but not in P and y is bound in P but not in Q. Of course, there is a way to escape the above restriction by making P and Q communicate, but that option is prohibited by the typing rules.

Wadler [146] uses classical linear logic rather than intuitionistic linear logic. He justifies the choice for "greater simplicity and symmetry." In terms of provability, the shift from intuitionistic to classical linear logic will not make any difference. However, in terms of proof nets, one difficulty looms. In classical multiplicative linear logic, adding the Amida edges to proof nets seems harder than the intuitionistic case because classical MLL proof nets have no direction on edges. After a path crosses an Amida edge, the author has no idea which direction the path should continue; one plausible solution is allowing multiple processes to track the paths in parallel and making these processes synchronize on Amida edges.

That said, Wadler's type system is similar to ours; our abbreviations are largely taken from Wadler's translations.

Giunti and Vasconcelos's Type System

Giunti and Vasconcelos [62] give a type system for the pi calculus with the type preservation theorem. Their type system is extremely similar to our type system. They say "the goal of this work is to equip types with a constructor able to denote the two ends of a same channel" [62, Introduction]. One of their typing rules

$$\frac{\Gamma, x\colon (S, \overline{S}) \vdash P}{\Gamma \vdash (\nu x)P}$$

is similar to a rule in Theorem 2.4.5

$$\frac{\mathcal{O} \mathrel{\|} \Gamma, x\colon\varphi^{\sim}, y\colon\overline{\varphi^{\sim}} \vdash t\colon\psi}{\mathcal{O} \mathrel{\|} \Gamma \vdash \nu x\colon\varphi^{\sim}.t[x_L/x][x_R/y]\colon\psi} \enspace .$$

In many cases, the Curry-Howard correspondence is followed from the logical world to the programming world: for already known logics, new lambda calculi are invented (e.g. [2, 117, 142]). However, in our case, it seems that we have just found a connection between an already known logic called Abelian logic and an already known typing discipline invented by Giunti and Vasconcelos [62]. It will be worthwhile to compare their system with our type system closely.

2.7.3 Join Calculus

The join calculus [51] is a process calculus, which is used as a basis for a language JoCaml [53]. An implicit typing system [52] allowed the join calculus to be successfully merged with an ML family language OCaml, but the type system does not guarantee determinacy or ensure that all messages are consumed.

2.7.4 Continuations

The eval-subst rule enables an evaluation $\bar{c}[C[cv]] \Downarrow v$, which reminds us of the call-with-current-continuation primitive [122] and shift/reset primitives [6, 40]. The appearance of these classical type system primitives is not surprising because Abelian logic validates $((p \multimap q) \multimap q) \multimap p$, which is a stronger form of the double negation elimination. Possibly we could use the technique of Asai and Kameyama [6] to analyze the Amida calculus with eval-subst rule.

2.7.5 Logic Programming

There are at least two ways to interpret logics computationally. One is proof reduction, which is represented by λ-calculi. The other is proof searching. We have investigated the Amida calculus, which embodies the proof reduction approach to the Amida axiom. Then what implication does The Amida axiom have in the proof searching approach? Let us cite an example from Kobayashi and Yonezawa [89, A.2]:

> Consumption of a message m by a process $m \multimap B$ is represented by the following deduction:
>
> $$(m \otimes (m \multimap B) \otimes C) \multimap (B \otimes C)$$
>
> where C can be considered as other processes and messages, or an environment.

Using the Amida axiom, the inverse

$$(B \otimes C) \multimap (m \otimes (m \multimap B)) \otimes C$$

is derivable. This suggests that the Amida axiom states that some computation is reversible in the context of proof searching. We suspect that this can be useful within the realm of reversible computation [134].

2.8 Discussion

2.8.1 Adding Agents, Recursions, ...

Our type system has a drawback of having only finitely many processes. We expect it straightforward to overcome this weak point by adding recursions in our type system. As a guidance we can take μMALL of Baelde [14].

It is tempting to add modalities representing agents and then study the relationship with the multiparty session types [17, 81]. For that, intuitionistic epistemic logic by Hirai [75, 76] will be useful.

In order to implement this lambda calculus as a programming language, we need linear types. If we seek a quick way of implementing our hyper-lambda calculus on top of an existing ML family programming language, we have to choose the existing language carefully. For example, OCaml and Haskell uses intuitionistic type system so if we use these languages naively, the Amida axiom makes the type system inconsistent and some communication primitives can be thrown away, leaving the peer deadlocked. The uniqueness type of the programming language Clean [113] does not allow contraction, but the uniqueness type is still not special enough because their type system allows weakening.

2.8.2 Cut-Elimination

In the Amida calculus, there is a hypersequent that can be derived with the cut rule but not without cut rules. For example, $X, Y \vdash X \mid \vdash Y$ can be derived with a cut rule application:

$$\text{Cut}\,\dfrac{\text{Sync}\,\dfrac{\text{Merge}\,\dfrac{\text{Ax}\,\dfrac{}{Y \vdash Y} \quad \multimap\text{R}\,\dfrac{\text{Ax}\,\dfrac{}{X \vdash X}}{\vdash X \multimap X}}{Y \vdash Y \mid \vdash X \multimap X}}{Y \vdash \underline{X \multimap X} \mid \vdash Y} \qquad \multimap\text{L}\,\dfrac{\text{Merge}\,\dfrac{\text{Ax}\,\dfrac{}{X \vdash X} \quad \text{Ax}\,\dfrac{}{X \vdash X}}{X \vdash X \mid X \vdash X}}{\underline{X \multimap X}, X \vdash X}}{X, Y \vdash X \mid \vdash Y}$$

However, the conclusion is not derivable without the cut rule (as confirmed by a simple proof searching).

Worse, it is easy to see that the prelinearity $(\varphi \multimap \psi) \oplus (\psi \multimap \varphi)$ does not have a cut-free proof. However, since there is a cut-free deduction system for Abelian logic [102], we consider it natural to expect the same property for a suitable extension of the Amida calculus, which would contain terms and reductions corresponding to proof rules and proof reductions in the system of Metcalfe [102].

2.8.3 The Problem with Excluded Middle

Since the Amida calculus characterizes Abelian logic, there is a closed term t with the sequent $\vdash t\colon (X \multimap \mathbf{1}) \oplus X$ derivable. We tried looking at one such term t, but it involved channels contained in lambda abstraction bodies and additive pairs. Unfortunately, the closed lambda term does not choose left or right so that it does not give a constructive justification to the excluded middle. Since the closed term t uses additive constructs, we expect the multiplicative fragment to be much more explainable.

2.8.4 Non-Commutative Version of the Amida Axiom

Including the the non-commutative version of the Amida axiom $(\varphi/\psi) \otimes (\psi/\varphi)$ to the full Lambek calculus turns out to be equivalent to adding $(\psi\backslash\varphi) \otimes (\varphi\backslash\psi)$. Algebraically speaking, the set of equations of the form $(x/y)\cdot(y/x) = 1$ forms an equational basis for LG, the variety of lattice-ordered groups. In a lattice-ordered group, $(x/y) \cdot (y/x) = (x \cdot y^{-1}) \cdot (y \cdot x^{-1}) = 1$. On the other hand, assuming $(x/y) \cdot (y/x) = 1$, when we substitute 1 for x, we obtain $(1/y) \cdot (y/1) = 1$. By the definition of residuation and the unit 1 of $\cdot$, we have $(1/y) \cdot y = 1$, which forms an equational basis for the variety of lattice-ordered groups [57, Lemma 3.25].

2.8.5 Program Specification and Verification

Since the typing derivations of the Amida calculus can be interpreted as proofs of Abelian logic, there is a possibility of specifying properties of processes using (first-order) Abelian logic formulae and then proving those properties using conjunctive hypersequents[14]. If that succeeds, the technique would be the first application of Abelian logic in program verification.

2.9 Conclusions

We found a new axiomatization of Abelian logic: the Amida axiom $(\varphi \multimap \psi)\otimes(\psi \multimap \varphi)$ on top of IMALL^{-}. The axiomatization has an application for encoding process calculi and session type systems. As a technique, we used conjunctive hypersequents for the first time, where components in a hypersequent are interpreted conjunctively rather than disjunctively. The resulting hyper-lambda calculus is somewhat hard to treat theoretically. If we need convergence or deadlock-freedom, we need the eval-subst rule in the evaluation rule (Subsection 2.3.1), which poses a threat to type safety. Moreover, there are some inhabited types like $\varphi \oplus (\varphi \multimap \mathbf{1})$ and $(\varphi \multimap \psi) \oplus (\psi \multimap \varphi)$, whose constructive justification is hard to see. Plausibly, the situation can be remedied by extending the Amida net approach treated in Section 2.6.

[14]When the author applied for Grant-in-Aid for JSPS Fellows 23-6978, the proposal included such a plan. According to the proposal, this research would proceed from finding a typed lambda calculus then to developing a logic for proving properties of the obtained programming language.

Chapter 3

An Asynchronous Hyper-Lambda Calculus

3.1 Summary

We define a typed lambda calculus based on Avron's hypersequent calculus [8] for Gödel-Dummett logic. We show that this calculus models waitfree computation [70, 125]. Besides strong normalization and non-abortfullness, we give soundness and completeness of the calculus against the typed version of waitfree protocols. The calculus is not only proof-theoretically interesting, but also valuable as a basis for distributed programming languages. In other words, we extend Curry-Howard isomorphism [130] to Gödel-Dummett logic and waitfreedom. The content of this chapter appeared in a conference paper by Hirai [77] although substantial modification has been applied since.

On one hand, Gödel-Dummett logic [43] is one of the intermediate logics between classical and intuitionistic logics. On the other hand, waitfreedom [70, 125] is a class of distributed computation without synchronization among processes.

We connect Gödel-Dummett logic and waitfreedom using Avron's hypersequent calculus [8]. In doing that, we respond to his suggestion:

> it seems to us extremely important to determine the exact computational content of them [intermediate logics]—and to develop corresponding 'λ-calculi' —Avron [8].

Differently from intuitionistic logic, Gödel-Dummett logic validates all formulae of the form $(\varphi \supset \psi) \vee (\psi \supset \varphi)$. We aim at building a typed lambda calculus with some terms witnessing those formulae. Such a term $M : (\varphi \supset \psi) \vee (\psi \supset \varphi)$ must choose $M \rightsquigarrow^* \mathsf{inl}(\cdots)$ or $M \rightsquigarrow^* \mathsf{inr}(\cdots)$. We devise a nondeterministic lambda calculus in Section 3.2. In this lambda calculus, terms can contain operators that read from and write to the store, or memory. In essence, the nondeterminism arises from the scheduling of concurrent processes: which process writes to the store before which process reads from the store.

Waitfreedom is a class of distributed computation where processes cannot wait for other processes. When two processes try to exchange information, the faster process can pass information to the slower one, but not always vice versa because the slower process might start after the faster one finishes. In this situation, computation is nondeterministic in general.

The contribution of this chapter lies in capturing the nondeterminism of waitfreedom using the nondeterministic λ-calculus for Gödel-Dummett logic. In Section 3.2, we define a lambda calculus and prove some of its properties. In Section 3.3, we define typed waitfreedom. Finally in Section 3.4, we prove that the lambda calculus characterizes typed waitfreedom. In other words, we show that the λ-terms in the lambda calculus can solve a typed input-output problem if and only if it is waitfreely solvable.

3.2 λ-GD

In this section, we define a typed lambda calculus λ-GD based on hypersequents. This constitutes the second contribution of this thesis. Moreover, the lambda calculus is one side of the Curry-Howard isomorphism that will be matched against the waitfree computation.

We first present a proof system for Gödel-Dummett logic. Then we turn the proof system into typing rules for λ-terms of λ-GD, give a set of reductions and prove strong-normalization and non-abortfullness. We show that the proof system using the hypersequent style [8]. In the usual sequent calculi, each reasoning step concludes a sequent $\Gamma \vdash \varphi$ where Γ is a possibly empty sequence of formulae. By contrast, in the hypersequent calculi, each reasoning step concludes a hypersequent, which is a finite, nonempty sequence of sequents. Of the hypersequent $\Gamma_0 \vdash \varphi_0 \mathbin{\|} \Gamma_1 \vdash \varphi_1 \mathbin{\|} \cdots \mathbin{\|} \Gamma_n \vdash \varphi_n$, each sequent $\Gamma_i \vdash \varphi_i$ is called a *component*. Each component $\Gamma_i \vdash \varphi_i$ is interpreted as an implication: the conjunction of Γ_i implies φ_i. The whole hypersequent is interpreted as disjunction of implications: $\vdash$ is interpreted as implication and $\|$ as disjunction. In other words, Γ_i implies φ_i for at least one $i \in \{0, 1, \ldots, n\}$. When we give computational interpretation to proofs, we still interpret the components disjunctively: namely, a derivation tree concluding a hypersequent represents a sequence of concurrent processes at least one of which is guaranteed to succeed.

3.2.1 Logic

Let us assume a countably infinite set PVar of *propositional variables*. We define *local formulae* φ, ψ (also called *local types*) by the following BNF, where I is a propositional variable[1]:

$$\varphi, \psi ::= \bot \mid I \mid (\varphi \supset \psi) \mid (\varphi \wedge \psi) \mid (\varphi \vee \psi) \ .$$

We omit parentheses when no ambiguity occurs. We assume a countably infinite number of *processes*. A *global formula* (also called a *global type*) is a non-empty partial map from processes to local formulae. We use φ^+ and ψ^+ for global formulae. For a process i, as a notation, $[i]\varphi$ is a global formula that maps i to φ but does not map any other processes to local formulae. We name such global formulae that map only one process as *singleton global formulae*. The unary operators $[0], [1], \ldots$ are called *modalities*. Informally, the local formulae describe datatypes used by each process. The global formulae describe inputs or outputs of possibly multiple processes.

A *context* (denoted by Γ and Δ possibly subscripted) is a potentially empty finite sequence of singleton global formulae. A *sequent* $\Gamma \vdash \varphi^+$ is a pair of a context and a global formula. A *hypersequent* is a finite sequence of sequents.

The underlying logic has the derivation rules in Figure 3.2. If we omit all the modalities, these rules characterize Gödel-Dummett logic.

Theorem 3.2.1 (Characterization of Gödel-Dummett logic) *For any local formula φ, φ is a theorem in Gödel-Dummett logic iff $[0]\varphi$ is provable.*

Proof We consider a translation of a global formula φ^+ to a local formula. Let I be the domain of φ^+. The translation is $\bigwedge_{i\in I} \varphi^+(i)$. After translating all rules in Figure 3.2, both directions of the statement can be shown by induction on derivations. ■

Indeed, $[0]((\varphi \supset \psi) \vee (\psi \supset \varphi))$ is provable (Figure 3.1).

3.2.2 Term Assignment

We assume distinct, countably infinite sets of *variables* and *locations*. Locations are denoted by $c, d, \ldots$; process $i, j, \ldots$ and variables $x, y, \ldots$. Later, locations will be used to specify a location in a store. A location in a store can hold a term or be empty. Like in the λ-calculus, some terms reduce to other terms, but in this calculus, terms may interact with stores (like a program written in Haskell or OCaml does with i-vars). This behavior will be shown later in the definition of reductions.

[1]We include $\bot$ in the definition of local formulae because Gödel-Dummett logic has $\bot$ although $\bot$ is not necessary to encode waitfree computation.

$$\text{EC}\,\dfrac{[0]\vee\mathcal{I}\,\dfrac{[0]\vee\mathcal{I}\,\dfrac{[0]\supset\mathcal{I}\,\dfrac{[0]\supset\mathcal{I}\,\dfrac{\text{00-com}\,\dfrac{[0]\text{Ax}\,\dfrac{}{[0]P\vdash[0]P}\qquad[0]\text{Ax}\,\dfrac{}{[0]Q\vdash[0]Q}}{[0]P\vdash[0]Q \mid [0]Q\vdash[0]P}}{\vdash[0](P\supset Q)\mid[0]Q\vdash[0]P}}{\vdash[0](P\supset Q)\mid\ \vdash[0](Q\supset P)}}{\vdash[0]((P\supset Q)\vee(Q\supset P))\mid\ \vdash[0](Q\supset P)}}{\vdash[0]((P\supset Q)\vee(Q\supset P))\mid\ \vdash[0](P\supset Q)\vee(Q\supset P)}}{\vdash[0]((P\supset Q)\vee(Q\supset P))}$$

Figure 3.1: A derivation of Dummett's axiom under modality [0] in λ-GD.

We define *local terms* M by BNF:

$$M ::= x \mid (*^{\to c}_{\leftarrow c})M \mid *_{\leftarrow c} \mid [M,M] \mid \mathsf{abort} \mid \pi_{\mathsf{l}}(M) \mid \pi_{\mathsf{r}}(M) \mid \langle M,M\rangle \mid$$
$$\mathsf{inl}(M) \mid \mathsf{inr}(M) \mid\mid \mathsf{match}\, M \,\mathsf{of}\, \mathsf{inl}(x).M/\mathsf{inr}(y).M\lambda x.M \mid (MM)$$

where x is a variable and c is a location. All variable occurrences except the first clause are binding. The set of free variables of a local term $\mathrm{FV}(M)$ is defined inductively on M:

$$\begin{aligned}
\mathrm{FV}(x) &= \{x\}\\
\mathrm{FV}(*^{\to c}_{\leftarrow c})M &= \mathrm{FV}(M)\\
\mathrm{FV}(*_{\leftarrow c}) &= \emptyset\\
\mathrm{FV}([M,N]) &= \mathrm{FV}(M)\cup\mathrm{FV}(N)\\
\mathrm{FV}(\mathsf{abort}\,) &= \emptyset\\
\mathrm{FV}(\pi_{\mathsf{l}}(M)) &= \mathrm{FV}(M)\\
\mathrm{FV}(\pi_{\mathsf{r}}(N)) &= \mathrm{FV}(N)\\
\mathrm{FV}(\langle M,N\rangle) &= \mathrm{FV}(M)\cup\mathrm{FV}(N)\\
\mathrm{FV}(\mathsf{inl}(M)) &= \mathrm{FV}(M)\\
\mathrm{FV}(\mathsf{inr}(M)) &= \mathrm{FV}(M)\\
\mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M)\setminus\{x\}\\
\mathrm{FV}((MN)) &= \mathrm{FV}(M)\cup\mathrm{FV}(N)\\
\mathrm{FV}(\mathsf{match}\,M\,\mathsf{of}\,\mathsf{inl}(x).N/\mathsf{inr}(y).O) &= \mathrm{FV}(M)\cup(\mathrm{FV}(N)\setminus\{x\})\cup(\mathrm{FV}(O)\setminus\{y\})\ .
\end{aligned}$$

A *closed local term* is a local term that has no free variables.

A *global term* is a partial map from processes to local terms. For example, a global term $\{0\mapsto\lambda x.x, 1\mapsto y\}$ maps process 0 to local term $\lambda x.x$ and process 1 to local term y. We can substitute a local term for a variable in a global term. The substitution is

External Rules

$$ij\text{-com}\ \frac{\mathcal{H}_0 \mid \Gamma \vdash [i]\varphi \qquad \mathcal{H}_1 \mid \Delta \vdash [j]\psi}{\mathcal{H}_0 \mid \mathcal{H}_1 \mid \Gamma \vdash [i]\psi \mid \Delta \vdash [j]\varphi} \qquad\qquad \text{EW}\ \frac{\mathcal{H}^+}{\mathcal{H}^+ \mid \Gamma \vdash \varphi^+}$$

$$\text{EC}\ \frac{\mathcal{H} \mid \Gamma \vdash \varphi^+ \mid \Gamma \vdash \varphi^+}{\mathcal{H} \mid \Gamma \vdash \varphi^+} \qquad\qquad \text{EE}\ \frac{\mathcal{H} \mid \Gamma \vdash \varphi^+ \mid \Delta \vdash \psi^+ \mid \mathcal{H}'}{\mathcal{H} \mid \Delta \vdash \psi^+ \mid \Gamma \vdash \varphi^+ \mid \mathcal{H}'}$$

Inner Global Rules

$$\text{IE}\ \frac{\mathcal{H} \mid \Gamma, [i]\varphi, [j]\psi, \Delta \vdash \theta^+}{\mathcal{H} \mid \Gamma, [j]\psi, [i]\varphi, \Delta \vdash \theta^+} \qquad \text{IW}\ \frac{\mathcal{H} \mid \Gamma \vdash \varphi^+}{\mathcal{H} \mid [i]\psi, \Gamma \vdash \varphi^+} \qquad \text{IC}\ \frac{\mathcal{H} \mid [i]\psi, [i]\psi, \Gamma \vdash \varphi^+}{\mathcal{H} \mid [i]\psi, \Gamma \vdash \varphi^+}$$

$$\wedge\mathcal{I}\ \frac{\mathcal{H}_0 \mid \Gamma, \Delta_0 \vdash (\varphi_i)_{i\in I} \qquad \mathcal{H}_1 \mid \Gamma, \Delta_1 \vdash (\psi_j)_{j\in J}}{\mathcal{H}_0 \mid \mathcal{H}_1 \mid \Gamma, \Delta_0, \Delta_1 \vdash (\varphi_k \wedge \psi_k)_{k\in I\cap J} \sqcup (\varphi_i)_{i\in I\setminus J} \sqcup (\psi_j)_{j\in J\setminus I}}$$

$$\wedge\mathcal{E}\ \frac{\mathcal{H} \mid \Gamma \vdash (\varphi_i)_{i\in I}}{\mathcal{H} \mid \Gamma \vdash (\varphi_i)_{i\in J}}\ \text{where } J \text{ is a subset of } I.$$

Inner Local Rules

$$[i]\text{Ax}\ \frac{}{[i]\varphi, \Gamma \vdash [i]\varphi} \qquad\qquad [i]\bot\mathcal{E}\ \frac{\mathcal{H} \mid \Gamma \vdash [i]\bot}{\mathcal{H} \mid \Gamma \vdash [i]\varphi}$$

$$[i]\supset\mathcal{I}\ \frac{\mathcal{H} \mid [i]\varphi, \Gamma \vdash [i]\psi}{\mathcal{H} \mid \Gamma \vdash [i](\varphi \supset \psi)} \qquad [i]\supset\mathcal{E}\ \frac{\mathcal{H}_0 \mid \Gamma, \Delta_0 \vdash [i](\varphi \supset \psi) \qquad \mathcal{H}_1 \mid \Gamma, \Delta_1 \vdash [i]\varphi}{\mathcal{H}_0 \mid \mathcal{H}_1 \mid \Gamma, \Delta_0, \Delta_1 \vdash [i]\psi}$$

$$[i]\wedge\mathcal{E}_0\ \frac{\mathcal{H} \mid \Gamma \vdash [i](\varphi \wedge \psi)}{\mathcal{H} \mid \Gamma \vdash [i]\varphi} \qquad\qquad [i]\wedge\mathcal{E}_1\ \frac{\mathcal{H} \mid \Gamma \vdash [i](\varphi \wedge \psi)}{\mathcal{H} \mid \Gamma \vdash [i]\psi}$$

$$[i]\vee\mathcal{I}_0\ \frac{\mathcal{H} \mid \Gamma \vdash [i]\varphi}{\mathcal{H} \mid \Gamma \vdash [i](\varphi \vee \psi)} \qquad\qquad [i]\vee\mathcal{I}_1\ \frac{\mathcal{H} \mid \Gamma \vdash [i]\psi}{\mathcal{H} \mid \Gamma \vdash [i](\varphi \vee \psi)}$$

$$[i]\vee\mathcal{E}\ \frac{\mathcal{H}_0 \mid \Gamma, \Delta_0 \vdash [i](\varphi \vee \psi) \qquad \mathcal{H}_1 \mid [i]\varphi, \Gamma, \Delta_1 \vdash [i]\theta \qquad \mathcal{H}_2 \mid [i]\psi, \Gamma, \Delta_2 \vdash [i]\theta}{\mathcal{H}_0 \mid \mathcal{H}_1 \mid \mathcal{H}_2 \mid \Gamma, \Delta_0, \Delta_1, \Delta_2 \vdash [i]\theta}$$

Figure 3.2: The underlying logic. Metavariables φ^+ and ψ^+ stand for global formulae. Metavariables i and j stand for processes. For the use of $\sqcup$, see Section 1.2. $\mathcal{H}$ stands for a possibly empty hypersequent. $\mathcal{H}^+$ stands for a nonempty hypersequent. Γ and Δ stand for possibly empty contexts. In the names of the rules, $\mathcal{I}$ at the end stands for introduction and $\mathcal{E}$ for elimination. For the structural rules, E stands for exchange, W for weakening and C for contraction. The I's in front stand for internal and E for external. A rule $[i]\wedge\mathcal{I}$ is omitted because the inner global rule $\wedge\mathcal{I}$ is more general. There is no disjunction elimination in the inner global rules lest it is difficult (if possible) to translate the rule into waitfree computation.

defined elementwise: i.e., the result of the substitution $M^+[N/x]$ maps a process i to $(M^+(i))[N/x]$. A *closed global term* is a global term whose image only contains closed local terms. Informally, the local terms represent parts of programs executed by the processes. Especially, $(*^{\rightarrow c}_{\leftarrow c})t$ and $*_{\leftarrow c}$ cause processes to communicate.

Using these terms, we annotate the hypersequent system in Figure 3.2. We extend a sequent to $\Gamma \vdash M^+ : \varphi^+$, where Γ is a sequence like $x : [i]\psi, y : [j]\theta$ and M^+ is a global term. In a sequent $\Gamma \vdash M^+ : \varphi^+$, we require the variables in Γ to be distinct from each other. We do not allow a context to contain both $x : [0]I$ and $x : [1]I$, or even $x : [0]I$ twice. An (extended) *hypersequent* is a finite sequence of sequents (each called a *component*) where the same variable has the same type even if it appears in different components. The typing rules for the terms are given in Figure 3.3. For example, the proof in Figure 3.1 can be annotated as in Figure 3.4. Each derivation has a unique hypersequent at the bottom, which is called the *endhypersequent* of the derivation. A hypersequent is *derivable* when there is a derivation whose endsequent is identical to the hypersequent.

Example 3.2.2 (A typing of a global term.) *The global term*

$$\{0 \mapsto [\langle (*^{\rightarrow d}_{\leftarrow c})x, x\rangle, x], 1 \mapsto [\langle (*^{\rightarrow c}_{\leftarrow d})y, y\rangle, y]\}$$

can be typed (Figure 3.5).

3.2.3 Reduction

A *hyperterm* $\mathcal{O}$ is a nonempty sequence of global terms. From a hypersequent $\Gamma_0 \vdash {M_0}^+ : {\varphi_0}^+ \mid \cdots \mid \Gamma_n \vdash {M_n}^+ : {\varphi_n}^+$, we can construct a hyperterm $({M_i}^+)_{0 \le i \le n}$. The hyperterm $({M_i}^+)_{0 \le i \le n}$ is typable when the aforementioned hypersequent is derivable for some Γ_i's and ${\varphi_i}^+$'s.

A *store* maps a location to a pure, closed term or ϵ, where a *pure term* is a term without $*^{\rightarrow c}_{\leftarrow c}$ or $*_{\leftarrow c}$. For a store S, the *updated store* $S[c \mapsto x]$ maps c to x and d to $S(d)$ if d is different from c. A *configuration* is a pair $(S, \mathcal{O})$ of a store S and a hyperterm $\mathcal{O}$. A *typed configuration* is a configuration $(\epsilon, \mathcal{O})$ where ϵ is the empty store and $\mathcal{O}$ is a typable hypersequent.

To complete the definition of λ-GD, we define the *reductions* $\rightsquigarrow_{\spadesuit}$ of configurations for $\spadesuit \in \{\mathrm{B}, \mathrm{W}, \mathrm{R}, \mathrm{A}, \mathrm{P}\}$, where B stands for basic, W for write, R for read, A for abort and P for permutative reductions. We consider terms up to α-equivalence and implicitly require all instances of $\rightsquigarrow_{\spadesuit}$ to avoid free variable captures.

$$\text{com}\ \dfrac{\mathcal{O}_0 \mid \Gamma \vdash \{i \mapsto M\} : [i]\varphi \qquad \mathcal{O}_1 \mid \Delta \vdash \{j \mapsto N\} : [j]\psi}{\mathcal{O}_0 \mid \mathcal{O}_1 \mid \Gamma \vdash \{i \mapsto (*\substack{\rightarrow d\\ \leftarrow c})M\} : [i]\psi \mid \Delta \vdash \{j \mapsto (*\substack{\rightarrow c\\ \leftarrow d})N\} : [j]\varphi}$$

where locations c and d are fresh and no location is contained in both of the two branches.

$$\text{EC}\ \dfrac{\mathcal{O} \mid \Gamma \vdash (M_i)_{i\in I} : (\varphi_i)_{i\in I} \mid \Gamma \vdash (N_i)_{i\in I} : (\varphi_i)_{i\in I}}{\mathcal{O} \mid \Gamma \vdash ([M_i, N_i])_{i\in I} : (\varphi_i)_{i\in I}}$$

$$\text{EW}\ \dfrac{\mathcal{O}^+}{\mathcal{O}^+ \mid \Gamma \vdash \{\} : \varphi^+} \qquad \text{EE}\ \dfrac{\mathcal{O} \mid \Gamma \vdash M^+ : \varphi^+ \mid \Delta \vdash N^+ : \psi^+ \mid \mathcal{O}'}{\mathcal{O} \mid \Delta \vdash N^+ : \psi^+ \mid \Gamma \vdash M^+ : \varphi^+ \mid \mathcal{O}'}$$

$$\text{IW}\ \dfrac{\mathcal{O} \mid \Gamma \vdash M^+ : \varphi^+}{\mathcal{O} \mid x : [i]\psi, \Gamma \vdash M^+ : \varphi^+} \qquad \text{IC}\ \dfrac{\mathcal{O} \mid x : [i]\varphi, y : [i]\varphi, \Gamma \vdash M^+ : \psi^+}{\mathcal{O} \mid x : [i]\varphi, \Gamma \vdash M^+[x/y] : \psi^+}$$

$$\text{IE}\ \dfrac{\mathcal{O} \mid \Gamma, x : [i]\varphi, y : [j]\psi, \Delta \vdash M^+ : \theta^+}{\mathcal{O} \mid \Gamma, y : [j]\psi, x : [i]\varphi, \Delta \vdash M^+ : \theta^+}$$

$$\wedge\mathcal{I}\ \dfrac{\mathcal{O}_0 \mid \Gamma, \Delta_0 \vdash (M_i)_{i\in I} : (\varphi_i)_{i\in I} \qquad \mathcal{O}_1 \mid \Gamma, \Delta_1 \vdash (N_j)_{j\in J} : (\psi_j)_{j\in J}}{\begin{array}{l}\mathcal{O}_0 \mid \mathcal{O}_1 \mid \Gamma, \Delta_0, \Delta_1 \vdash \\ (\langle M_k, N_k\rangle)_{k\in I\cap J} \sqcup (M_i)_{i\in I\setminus J} \sqcup (N_j)_{j\in J\setminus I} : (\varphi_k \wedge \psi_k)_{k\in I\cap J} \sqcup (\varphi_i)_{i\in I\setminus J} \sqcup (\psi_j)_{j\in J\setminus I}\end{array}}$$

$$\wedge\mathcal{E}\ \dfrac{\mathcal{O} \mid \Gamma \vdash (M_i)_{i\in I} : (\varphi_i)_{i\in I}}{\mathcal{O} \mid \Gamma \vdash (M_j)_{j\in J} : (\varphi_j)_{j\in J}} \text{ where } J \text{ is a subset of } I.$$

$$[i]\text{Ax}\ \dfrac{}{x : [i]\varphi, \Gamma \vdash \{i \mapsto x\} : [i]\varphi} \qquad [i]\bot\mathcal{E}\ \dfrac{\mathcal{O} \mid \Gamma \vdash \{i \mapsto M\} : [i]\bot}{\mathcal{O} \mid \Gamma \vdash \{i \mapsto \mathsf{abort}\} : [i]\varphi}$$

$$[i] \supset \mathcal{I}\ \dfrac{\mathcal{O} \mid x : [i]\varphi, \Gamma \vdash \{i \mapsto M\} : [i]\psi}{\mathcal{O} \mid \Gamma \vdash \{i \mapsto \lambda x.M\} : [i](\varphi \supset \psi)}$$

$$[i] \supset \mathcal{E}\ \dfrac{\mathcal{O}_0 \mid \Gamma, \Delta_0 \vdash \{i \mapsto M\} : [i](\varphi \supset \psi) \qquad \mathcal{O}_1 \mid \Gamma, \Delta_1 \vdash \{i \mapsto N\} : [i]\varphi}{\mathcal{O}_0 \mid \mathcal{O}_1 \mid \Gamma, \Delta_0, \Delta_1 \vdash \{i \mapsto MN\} : [i]\psi}$$

$$[i] \wedge \mathcal{E}_0\ \dfrac{\mathcal{O} \mid \Gamma \vdash \{i \mapsto M\} : [i](\varphi \wedge \psi)}{\mathcal{O} \mid \Gamma \vdash \{i \mapsto \pi_{\mathsf{l}}(M)\} : [i]\varphi} \qquad [i] \wedge \mathcal{E}_1\ \dfrac{\mathcal{O} \mid \Gamma \vdash \{i \mapsto M\} : [i](\varphi \wedge \psi)}{\mathcal{O} \mid \Gamma \vdash \{i \mapsto \pi_{\mathsf{r}}(M)\} : [i]\psi}$$

$$[i] \vee \mathcal{I}_0\ \dfrac{\mathcal{O} \mid \Gamma \vdash \{i \mapsto M\} : [i]\varphi}{\mathcal{O} \mid \Gamma \vdash \{i \mapsto \mathsf{inl}(M)\} : [i](\varphi \vee \psi)} \qquad [i] \vee \mathcal{I}_1\ \dfrac{\mathcal{O} \mid \Gamma \vdash \{i \mapsto M\} : [i]\psi}{\mathcal{O} \mid \Gamma \vdash \{i \mapsto \mathsf{inr}(M)\} : [i](\varphi \vee \psi)}$$

$$[i] \vee \mathcal{E}\ \dfrac{\begin{array}{c}\mathcal{O}_0 \mid \Gamma, \Delta_0 \vdash \{i \mapsto M\} : [i](\varphi \vee \psi) \\ \mathcal{O}_1 \mid x : [i]\varphi, \Gamma, \Delta_1 \vdash \{i \mapsto N_0\} : [i]\theta \\ \mathcal{O}_2 \mid y : [i]\psi, \Gamma, \Delta_2 \vdash \{i \mapsto N_1\} : [i]\theta\end{array}}{\mathcal{O}_0 \mid \mathcal{O}_1 \mid \mathcal{O}_2 \mid \Gamma, \Delta_0, \Delta_1, \Delta_2 \vdash \{i \mapsto \mathsf{match}\, M\, \mathsf{of}\, \mathsf{inl}(x).N_0/\mathsf{inr}(y).N_1\} : [i]\theta}$$

Figure 3.3: Term assignment on the λ-GD type system. Metavariable M^+ stands for global terms. $\mathcal{O}$ stands for a possibly empty hypersequent (with possible subscripts). $\mathcal{O}^+$ stands for a non-empty hypersequent.

$$\mathrm{EC}\,\dfrac{[0]\vee\mathcal{I}\,\dfrac{[0]\vee\mathcal{I}\,\dfrac{[0]\supset\mathcal{I}\,\dfrac{[0]\supset\mathcal{I}\,\dfrac{\text{00-com}\,\dfrac{[0]\mathrm{Ax}\,\dfrac{}{x:[0]P\vdash x:[0]P}\qquad [0]\mathrm{Ax}\,\dfrac{}{y:[0]Q\vdash y:[0]Q}}{x:[0]P\vdash(*^{\rightarrow d}_{\leftarrow c})x:[0]Q\;\rule[-0.2em]{0.2em}{1em}\;y:[0]Q\vdash(*^{\rightarrow c}_{\leftarrow d})y:[0]P}}{\vdash\lambda x.(*^{\rightarrow d}_{\leftarrow c})x:[0](P\supset Q)\;\rule[-0.2em]{0.2em}{1em}\;y:[0]Q\vdash(*^{\rightarrow c}_{\leftarrow d})y:[0]P}}{\vdash\lambda x.(*^{\rightarrow d}_{\leftarrow c})x:[0](P\supset Q)\;\rule[-0.2em]{0.2em}{1em}\;\vdash\lambda y.(*^{\rightarrow c}_{\leftarrow d})y:[0](Q\supset P)}}{\vdash\mathsf{inl}(\lambda x.(*^{\rightarrow d}_{\leftarrow c})x):[0]((P\supset Q)\vee(Q\supset P))\;\rule[-0.2em]{0.2em}{1em}\;\vdash\lambda y.(*^{\rightarrow c}_{\leftarrow d})y:[0](Q\supset P)}}{\vdash\mathsf{inl}(\lambda x.(*^{\rightarrow d}_{\leftarrow c})x):[0]((P\supset Q)\vee(Q\supset P))\;\rule[-0.2em]{0.2em}{1em}\;\vdash\mathsf{inr}(\lambda y.(*^{\rightarrow c}_{\leftarrow d})y):[0]((P\supset Q)\vee(Q\supset P))}}{\vdash\big[\mathsf{inl}(\lambda x.(*^{\rightarrow d}_{\leftarrow c})x),\quad\mathsf{inr}(\lambda y.(*^{\rightarrow c}_{\leftarrow d})y)\big]:[0]((P\supset Q)\vee(Q\supset P))}$$

Figure 3.4: An example of a typed term in λ-GD that corresponds the derivation in Figure 3.1.

Part A

$$[1]\vee\mathcal{I}\,\dfrac{[0]\vee\mathcal{I}\,\dfrac{\wedge\mathcal{I}\,\dfrac{\wedge\mathcal{I}\,\dfrac{\text{Ax}\,\dfrac{}{x:[0]X \vdash x:[0]X} \qquad \vdots}{x:[0]X \vdash \{0 \mapsto \langle(*^{\rightarrow d}_{\leftarrow c})x, x\rangle\}:[0](Y \wedge X) \;\rule{3pt}{10pt}\; y:[1]Y \vdash \{1 \mapsto (*^{\rightarrow c}_{\leftarrow d})y\}:[1]X} \qquad \text{Ax}\,\dfrac{}{y:[1]Y \vdash y:[1]Y}}{x:[0]X \vdash \{0 \mapsto \langle(*^{\rightarrow d}_{\leftarrow c})x, x\rangle\}:[0](Y \wedge X) \;\rule{3pt}{10pt}\; y:[1]Y \vdash \{1 \mapsto \langle(*^{\rightarrow c}_{\leftarrow d})y, y\rangle\}:[1](X \wedge Y)}}{x:[0]X \vdash \{0 \mapsto \mathsf{inl}\left(\langle(*^{\rightarrow d}_{\leftarrow c})x, x\rangle\right)\}:[0](Y \wedge X) \vee X \;\rule{3pt}{10pt}\; y:[1]Y \vdash \{1 \mapsto \langle(*^{\rightarrow c}_{\leftarrow d})y, y\rangle\}:[1](X \wedge Y)}}{x:[0]X \vdash \{0 \mapsto \mathsf{inl}\left(\langle(*^{\rightarrow d}_{\leftarrow c})x, x\rangle\right)\}:[0](Y \wedge X) \vee X \;\rule{3pt}{10pt}\; y:[1]Y \vdash \{1 \mapsto \mathsf{inr}(\langle(*^{\rightarrow c}_{\leftarrow d})y, y\rangle)\}:[1](X \wedge Y) \vee Y}$$

Part B

$$\wedge\mathcal{I}\,\dfrac{\wedge\mathcal{I}\,\dfrac{[1]\vee\mathcal{I}\,\dfrac{\text{Ax}\,\dfrac{}{y:[1]Y \vdash y:[1]Y}}{y:[1]Y \vdash \mathsf{inr}(y):[1]((X \wedge Y) \vee Y)} \qquad \vdots}{\begin{array}{c} x:[0]X \vdash \{0 \mapsto \mathsf{inl}\left(\langle(*^{\rightarrow d}_{\leftarrow c})x, x\rangle\right), 1 \mapsto \mathsf{inr}(y)\}:\{0 \mapsto (Y \wedge X) \vee X, 1 \mapsto (X \wedge Y) \vee Y\} \;\rule{3pt}{10pt} \\ y:[1]Y \vdash \{1 \mapsto \mathsf{inr}(\langle(*^{\rightarrow c}_{\leftarrow d})y, y\rangle)\}:[1](X \wedge Y) \vee Y \end{array}} \qquad [0]\vee\mathcal{I}\,\dfrac{\text{Ax}\,\dfrac{}{x:[0]X \vdash x:[0]X}}{\mathsf{inr}(x):[0]X \vdash x:[0]((Y \wedge X) \vee X)}}{\begin{array}{c} x:[0]X, y:[1]Y \vdash \{0 \mapsto \mathsf{inl}(\langle(*^{\rightarrow d}_{\leftarrow c})x, x\rangle), 1 \mapsto \mathsf{inr}(y)\}:\{0 \mapsto (Y \wedge X) \vee X, 1 \mapsto (X \wedge Y) \vee Y\} \;\rule{3pt}{10pt} \\ x:[0]X, y:[1]Y \vdash \{0 \mapsto \mathsf{inr}(x), 1 \mapsto \mathsf{inl}(\langle(*^{\rightarrow c}_{\leftarrow d})y, y\rangle)\}:\{0 \mapsto (Y \wedge X) \vee X, 1 \mapsto (X \wedge Y) \vee Y\} \end{array}}$$

Whole Derivation

$$\text{EC}\,\dfrac{\text{Part B}\,\dfrac{\text{Part A}\,\dfrac{\text{01-com}\,\dfrac{\text{Ax}\,\dfrac{}{x:[0]X \vdash \{0 \mapsto x\}:[0]X} \qquad \text{Ax}\,\dfrac{}{y:[1]Y \vdash \{1 \mapsto y\}:[1]Y}}{x:[0]X \vdash \{0 \mapsto (*^{\rightarrow d}_{\leftarrow c})x\}:[0]Y \;\rule{3pt}{10pt}\; y:[1]Y \vdash \{1 \mapsto (*^{\rightarrow c}_{\leftarrow d})y\}:[1]X}}{x:[0]X \vdash \{0 \mapsto \mathsf{inl}(\langle(*^{\rightarrow d}_{\leftarrow c})x, x\rangle)\}:[0]((Y \wedge X) \vee X) \;\rule{3pt}{10pt}\; y:[1]Y \vdash \{1 \mapsto \mathsf{inl}(\langle(*^{\rightarrow c}_{\leftarrow d})y, y\rangle)\}:[1]((X \wedge Y) \vee Y)}}{\begin{array}{c} x:[0]X, y:[1]Y \vdash \{0 \mapsto \mathsf{inl}(\langle(*^{\rightarrow d}_{\leftarrow c})x, x\rangle), 1 \mapsto \mathsf{inr}(y)\}:\{0 \mapsto (Y \wedge X) \vee X, 1 \mapsto (X \wedge Y) \vee Y\} \;\rule{3pt}{10pt} \\ x:[0]X, y:[1]Y \vdash \{0 \mapsto \mathsf{inr}(x), 1 \mapsto \mathsf{inl}(\langle(*^{\rightarrow c}_{\leftarrow d})y, y\rangle)\}:\{0 \mapsto (Y \wedge X) \vee X, 1 \mapsto (X \wedge Y) \vee Y\} \end{array}}}{x:[0]X, y:[1]Y \vdash \{0 \mapsto [\mathsf{inl}(\langle(*^{\rightarrow d}_{\leftarrow c})x, x\rangle), \mathsf{inr}(x)], 1 \mapsto [\mathsf{inl}(\langle(*^{\rightarrow c}_{\leftarrow d})y, y\rangle), \mathsf{inr}(y)]\}:\{0 \mapsto ((Y \wedge X) \vee X), 1 \mapsto ((X \wedge Y) \vee Y)\}}$$

Figure 3.5: An example of a typing derivation.

We require all reductions to be congruences. A binary relation R on configurations is a *congruence* when

- $(S_0, \mathcal{O}_0) R (S_1, \mathcal{O}_1)$ implies
  $(S_0, \mathcal{O}_0 \mid \mathcal{O}) R (S_1, \mathcal{O}_1 \mid \mathcal{O})$,
- $(S_0, \mathcal{O}_0 \mid {M_0}^+) R (S_1, \mathcal{O}_1 \mid {M_1}^+)$ implies
  $(S_0, \mathcal{O}_0 \mid {M_0}^+ \sqcup N^+) R (S_1, \mathcal{O}_1 \mid {M_1}^+ \sqcup N^+)$
- $(S_0, \mathcal{O}_0 \mid \{i \mapsto M_0\} \sqcup {M_0}^+) R (S_1, \mathcal{O}_1 \mid \{i \mapsto M_1\} \sqcup {M_1}^+)$ implies
  $(S_0, \mathcal{O}_0 \mid \{i \mapsto C[M_0]\} \sqcup {M_0}^+) R (S_1, \mathcal{O}_1 \mid \{i \mapsto C[M_1]\} \sqcup {M_1}^+)$ for any context C
- $(S, \mathcal{O}_0 \mid M^+ \mid N^+ \mid \mathcal{O}_1) R (S', \mathcal{O}')$ implies
  $(S, \mathcal{O}_0 \mid N^+ \mid M^+ \mid \mathcal{O}_1) R (S', \mathcal{O}')$ and
- $(S, \mathcal{O}) R (S', \mathcal{O}_0 \mid M^+ \mid N^+ \mid \mathcal{O}_1)$ implies
  $(S, \mathcal{O}) R (S', \mathcal{O}_0 \mid N^+ \mid M^+ \mid \mathcal{O}_1)$.

Although only singleton terms appear in these clauses, by congruence, these rules are applicable to more complicated global terms.

The first kind of reductions, basic reductions, are what one would expect from typed lambda calculi based on intuitionistic propositional logic with implication, conjunction and disjunction.

Definition 3.2.3 (Basic reduction) *The basic reduction* $\rightsquigarrow_{\mathrm{B}}$ *is the smallest congruence containing the followings:*

- $(S, \{i \mapsto (\lambda x.M)N\}) \rightsquigarrow_{\mathrm{B}} (S, \{i \mapsto M[N/x]\})$
- $(S, \{i \mapsto \pi_{\mathsf{l}}(\langle M, N\rangle)\}) \rightsquigarrow_{\mathrm{B}} (S, \{i \mapsto M\})$
- $(S, \{i \mapsto \pi_{\mathsf{r}}(\langle M, N\rangle)\}) \rightsquigarrow_{\mathrm{B}} (S, \{i \mapsto N\})$
- $(S, \{i \mapsto \mathsf{match}\,(\mathsf{inl}\,(M))\,\mathsf{of}\,\mathsf{inl}(x).N/\mathsf{inr}(y).O\}) \rightsquigarrow_{\mathrm{B}} (S, \{i \mapsto N[M/x]\})$
- $(S, \{i \mapsto \mathsf{match}\,(\mathsf{inr}\,(M))\,\mathsf{of}\,\mathsf{inl}(x).N/\mathsf{inr}(y).O\}) \rightsquigarrow_{\mathrm{B}} (S, \{i \mapsto O[M/y]\})$

There are two sorts of reductions that interact with the store. In summary, $*\overset{\rightarrow d}{\underset{\leftarrow c}{}}$ tries to write to d and read from c in the store of the configuration. If a term writes to an empty location, the location is filled with the local term written by the term. If a term writes to a full location of a store, it does not abort but the store is not updated. In fact, the contents of locations are never updated after being written.

This property will be used in the proof of Theorem 3.2.11 (Strong normalization). The formal definition of the reductions follows.

Definition 3.2.4 (Write reduction) *The write reduction* $\rightsquigarrow_{\mathrm{W}}$ *is the smallest congruence containing the followings:*

- $(S[c \mapsto \epsilon], \{i \mapsto (*^{\to c}_{\leftarrow d})M\}) \rightsquigarrow_{\mathrm{W}} (S[c \mapsto M], \{i \mapsto *_{\leftarrow d}\})$ *where* M *is a pure, closed term*
- $(S[c \mapsto N], \{i \mapsto (*^{\to c}_{\leftarrow d})M\}) \rightsquigarrow_{\mathrm{W}} (S[c \mapsto N], \{i \mapsto *_{\leftarrow d}\})$ *where* M *is a pure, closed term.*

In the first clause of this definition of write reductions, we require M to be a pure, closed term because a store can only contain pure, closed terms. This restriction on store contents comes from our aim of modeling asynchronous communication. If we allow stores to contain terms with free variables, the term $(\lambda x.(*^{\to d}_{\leftarrow c}x))M$ can store x in the store and another process can read x from the store. After that, if $(\lambda x. \cdots)M$ reduces, then, x becomes M suddenly in the other process. We want to avoid such synchronous communication.

We require the same condition in the second clause as well although the store contents are not changed in the second clause. Otherwise, in order to judge whether a configuration admits a write reduction or not, we have to inspect the contents of the store. This would make the implementation more complicated. More technically, the definition of normal forms would depend on the store contents, which we avoided.

After the communicating term $(*^{\to d}_{\leftarrow c})$ writes to the memory, the term changes into a reader $*_{\leftarrow c}$. When the reader $*_{\leftarrow c}$ tries to read when c is full, the reader is replaced with the content of the location c. If a reader tries to read from an empty location of a store, the reader changes into abort.

Definition 3.2.5 (Read reduction) *The read reduction* $\rightsquigarrow_{\mathrm{R}}$ *is the smallest congruence containing the followings:*

- $(S[c \mapsto M], \{i \mapsto *_{\leftarrow c}\}) \rightsquigarrow_{\mathrm{R}} (S[c \mapsto M], \{i \mapsto M\})$
- $(S[c \mapsto \epsilon], \{i \mapsto *_{\leftarrow c}\}) \rightsquigarrow_{\mathrm{R}} (S[c \mapsto \epsilon], \{i \mapsto \mathsf{abort}\})$

The special term abort means failure, so, a term containing abort, except $[M, N]$, also reduces to abort. The *concurrent construction* $[M, N]$ runs M and N concurrently and throws away those subterms that reduce to abort. To be specific, the term $[M, N]$ reduces to M or N. The reduction rules are not symmetric with regard to the left component and the right component of the concurrent construct $[M, N]$ because when both components succeed, the whole construct reduces to the left component.

Definition 3.2.6 (Abort propagation reduction) *The abort propagation reduction $\rightsquigarrow_{\mathrm{A}}$ is the smallest congruence containing the followings:*

- $(S, \{i \mapsto [\mathsf{abort}\,, M]\}) \rightsquigarrow_{\mathrm{A}} (S, \{i \mapsto M\})$*;*
- $(S, \{i \mapsto [M, \mathsf{abort}\,]\}) \rightsquigarrow_{\mathrm{A}} (S, \{i \mapsto M\})$*;*
- $(S, \{i \mapsto [M, N]\}) \rightsquigarrow_{\mathrm{A}} (S, \{i \mapsto M\})$ *where M and N do not contain* abort*,* $*^{\to d}_{\leftarrow c}$ *or $*_{\leftarrow c}$ for any locations c, d;*
- $(S, \{i \mapsto C[\mathsf{abort}\,]\}) \rightsquigarrow_{\mathrm{A}} (S, \{i \mapsto \mathsf{abort}\,\})$ *where $C[\bullet]$ is defined by BNF:*

$$\begin{aligned} C[\bullet] ::= {} & \bullet \mid C[\bullet]N \mid MC[\bullet] \mid (*^{\to c}_{\leftarrow c})C[\bullet] \mid \mathsf{inl}\,(C[\bullet]) \mid \mathsf{inr}\,(C[\bullet]) \mid \langle C[\bullet], N\rangle \mid \\ & \langle M, C[\bullet]\rangle \mid \pi_i^{\Box} C[\bullet] \mid \mathsf{match}\, M\, \mathsf{of}\ \mathsf{inl}(x).N/\mathsf{inr}(y).C[\bullet] \mid \\ & \mathsf{match}\, C[\bullet]\, \mathsf{of}\ \mathsf{inl}(x).N/\mathsf{inr}(y).O \mid \mathsf{match}\, M\, \mathsf{of}\ \mathsf{inl}(x).C[\bullet]/\mathsf{inr}(y).O \end{aligned}$$

For example, the configuration $(S, [\mathsf{abort}\,, \mathsf{abort}\,])$ reduces to $(S, \mathsf{abort}\,)$.

In order to avoid the situation where computation is blocked by a syntactic barricade, we add yet another kind of reduction rules called permutative reductions[2].

Definition 3.2.7 (Permutative reduction) *The permutative reduction $\rightsquigarrow_{\mathrm{P}}$ is the smallest congruence containing the followings:*

- $(S, \{i \mapsto (\mathsf{match}\, M\, \mathsf{of}\ \mathsf{inl}(x).N/\mathsf{inr}(y).O)\ P\}) \rightsquigarrow_{\mathrm{P}}$
  $(S, \{i \mapsto \mathsf{match}\, M\, \mathsf{of}\ \mathsf{inl}(x).NP/\mathsf{inr}(y).OP\})$
- $(S, \{i \mapsto \pi_d\,(\mathsf{match}\, M\, \mathsf{of}\ \mathsf{inl}(x).N/\mathsf{inr}(y).O)\}) \rightsquigarrow_{\mathrm{P}}$
  $(S, \{i \mapsto \mathsf{match}\, M\, \mathsf{of}\ \mathsf{inl}(x).\pi_d N/\mathsf{inr}(y).\pi_d O\})$
- $(S, \{i \mapsto \mathsf{match}\,(\mathsf{match}\, M\, \mathsf{of}\ \mathsf{inl}(x).N/\mathsf{inr}(y).O)\ \mathsf{of}\ \mathsf{inl}(u).P/\mathsf{inr}(v).Q\}) \rightsquigarrow_{\mathrm{P}}$
  $(S, \{i \mapsto \mathsf{match}\, M\, \mathsf{of}\ \mathsf{inl}\,(x)\,.(\mathsf{match}\, N\, \mathsf{of}\ \mathsf{inl}(u).P/\mathsf{inr}(v).Q)/$
  $\mathsf{inr}\,(y)\,.(\mathsf{match}\, O\, \mathsf{of}\ \mathsf{inl}(u).P/\mathsf{inr}(v).Q)\})$
- $(S, \{i \mapsto [M, N]P\}) \rightsquigarrow_{\mathrm{P}} (S, \{i \mapsto [MP, NP]\})$
- $(S, \{i \mapsto \pi_d[M, N]\}) \rightsquigarrow_{\mathrm{P}} (S, \{i \mapsto [\pi_d M, \pi_d N]\})$
- $(S, \{i \mapsto \mathsf{match}\, [M, N]\, \mathsf{of}\ \mathsf{inl}(x).P/\mathsf{inr}(y).Q\}) \rightsquigarrow_{\mathrm{P}}$
  $(S, \{i \mapsto [\mathsf{match}\, M\, \mathsf{of}\ \mathsf{inl}(x).P/\mathsf{inr}(y).Q, \mathsf{match}\, N\, \mathsf{of}\ \mathsf{inl}(x).P/\mathsf{inr}(y).Q]\})$

[2]The "permutative" conversions can be found in Prawitz [119].

We define $\rightsquigarrow$ to be the union of $\rightsquigarrow_{\mathrm{B}}$, $\rightsquigarrow_{\mathrm{W}}$, $\rightsquigarrow_{\mathrm{R}}$, $\rightsquigarrow_{\mathrm{A}}$ and $\rightsquigarrow_{\mathrm{P}}$. The reflexive transitive closure of $\rightsquigarrow$ is written as $\rightsquigarrow^*$. A *redex* is a subterm that can be rewritten by a reduction. A configuration $\mathcal{C}$ is *normal* when there is no configuration $\mathcal{D}$ with $\mathcal{C} \rightsquigarrow \mathcal{D}$. A term M is *normal* when the configuration (S, M) is a normal configuration (the choice of S is irrelevant).

Example 3.2.8 (Reductions of λ-GD) *We take the term shown in Figure 3.5 and replace x and y with pure, closed terms v and w. Below, an underlined subterm is a redex:*

$$\begin{aligned}
&\Big(\{\}, \{0 \mapsto [\langle \underline{(*\substack{\rightarrow d\\ \leftarrow c})v}, v\rangle, v], 1 \mapsto [\langle (*\substack{\rightarrow c\\ \leftarrow d})w, w\rangle, w]\Big) \\
\rightsquigarrow_{\mathrm{W}}\ &(\{d \mapsto v\}, \{0 \mapsto [\langle \underline{*_{\leftarrow c}}, v\rangle, v], 1 \mapsto [\langle (*\substack{\rightarrow c\\ \leftarrow d})w, w\rangle, w]\}) \\
\rightsquigarrow_{\mathrm{R}}\ &\Big(\{d \mapsto v\}, \{0 \mapsto [\underline{\langle \mathsf{abort}\,, v\rangle}, v], 1 \mapsto [\langle (*\substack{\rightarrow c\\ \leftarrow d})w, w\rangle, w]\}\Big) \\
\rightsquigarrow_{\mathrm{A}}\ &\Big(\{d \mapsto v\}, \{0 \mapsto \underline{[\mathsf{abort}\,, v]}, 1 \mapsto [\langle (*\substack{\rightarrow d\\ \leftarrow c})w, w\rangle, w]\}\Big) \\
\rightsquigarrow_{\mathrm{A}}\ &\Big(\{d \mapsto v\}, \{0 \mapsto v, 1 \mapsto [\langle \underline{(*\substack{\rightarrow c\\ \leftarrow d})w}, w\rangle, w]\}\Big) \\
\rightsquigarrow_{\mathrm{W}}\ &(\{c \mapsto w, d \mapsto v\}, \{0 \mapsto v, 1 \mapsto [\langle \underline{*_{\leftarrow d}}, w\rangle, w]\}) \\
\rightsquigarrow_{\mathrm{R}}\ &\Big(\{c \mapsto w, d \mapsto v\}, \{0 \mapsto v, 1 \mapsto \underline{[\langle v, w\rangle, w]}\}\Big) \\
\rightsquigarrow_{\mathrm{A}}\ &(\{c \mapsto w, d \mapsto v\}, \{0 \mapsto v, 1 \mapsto \langle v, w\rangle\}) \quad .
\end{aligned}$$

The overall effect is the transfer of term v from process 0 to process 1. In the beginning, v only appears in process 0's local term but not in process 1's local term. In the end, the term v appears in both processes' local terms. From the first line to the second line, process 0's communication term succeeds in writing term v to location d. From the second line to the third line, process 0 fails to read from location c so that a reader construct is turned into an abort . *This failure occurs because process 0 tries to read before process 1 writes. In a different scheduling (or, evaluation strategy) such a failure can be avoided. This kind of nondeterminism is essential to characterize waitfreedom later. From the third line to the fourth line, the* abort *in process 0's subterm is propagated so that the whole pair containing* abort *is turned into* abort . *From the fourth line to the fifth line, the aborted subterm in process 0 is thrown away: this is the main functionality of the concurrent construction $[M, N]$. From the fifth line to the sixth line, process 1 writes to location c. From the sixth line to the seventh line, process 1 reads what process 0 wrote. In this case, the reader in process 1 does not yield* abort *but outputs a previous content of the store. From the seventh line to the last line, the right side component w of $[\langle v, w\rangle, w]$ is thrown away: we chose to throw away the right hand side component when both sides of $[M, N]$ contain no* abort . *When* *we view the concurrent construction $[M, N]$ as a representation of external contraction rule, this asymmetry is strange, but the asymmetry plays a significant role in encoding waitfree computation in our calculus.*

3.2.4 Properties

An important property of λ-GD is strong normalization: every typed hyperterm has a finite, maximum number of ensuing reductions. Another property is non-abortfullness: although some reductions yield abort terms, a typed hyperterm never reduces to a hyperterm that only contains abort's. We show the second property first because its proof is simpler.

Theorem 3.2.9 (Non-abortfullness of $\boldsymbol{\lambda}$-GD) *All normal forms of a typed configuration contain at least one term that is not* abort*.*

Proof When a reduction sequence is fixed, for any locations c and d, depending on whether c or d is filled first, either: (i) no $*_{\leftarrow d} \rightsquigarrow \mathsf{abort}$ occurs, or (ii) no $*_{\leftarrow c} \rightsquigarrow \mathsf{abort}$ occurs.

In case (i), we can rewrite a communication rule occurrence

$$\frac{\mathcal{O}_0 \mid \Gamma, \Delta \vdash \{i \mapsto M\} : [i]\psi \qquad \mathcal{O}_1 \mid \Gamma, \Delta \vdash \{j \mapsto N\} : [j]\tau}{\mathcal{O}_0 \mid \mathcal{O}_1 \mid \Gamma \vdash \{i \mapsto (*^{\rightarrow d}_{\leftarrow c})M\} : [i]\tau \mid \Delta \vdash \{j \mapsto (*^{\rightarrow c}_{\leftarrow d})N\} : [j]\psi}$$

into a weakening occurrence (using Proposition 3.2.10 proved below)

$$\frac{\mathcal{O}_0 \mid \Gamma^j, \Delta^j \vdash \{j \mapsto M\} : [j]\psi}{\mathcal{O}_0 \mid \; \vdash \{i \mapsto \mathsf{abort}\} : [i]\tau \mid \Gamma^j, \Delta^j \vdash \{j \mapsto M\} : [j]\psi}$$

where Γ^j denotes the context obtained by replacing every modality in Γ with j. In case (ii), we can do the symmetric.

After these rewritings for all appearing locations, we obtain a derivation not containing any locations. Moreover, the endhypersequent of the rewritten derivation has a component not containing abort. The reductions of the original hyperterm can be simulated by the rewritten hyperterm. And, even after reductions, the resulting hyperterm has a component not containing abort. ■

In order to justify the renaming operation appearing in the proof of Theorem 3.2.9, we first define the renaming operation on global types. Of a global type $\varphi^{+} = (\varphi_i)_{i \in I}$ (the global type that maps i to φ_i), the j-replacement φ^{+j} is $\{j \mapsto \bigwedge_{i \in I}(\varphi_i)\}$. For example, the 1-replacement of $[0]\bot$ is $[0]\bot^1 = [1]\bot$.

For a context $\Gamma = x_0 : [i_0]\varphi_0, x_1 : [i_1]\varphi_1, x_2 : [i_2]\varphi_2, \ldots, x_n : [i_n]\varphi_n$, the j-replacement Γ^j is $x_0 : [j]\varphi_0, x_1 : [j]\varphi_1, x_2 : [j]\varphi_2, \ldots, x_n : [j]\varphi_n$.

Proposition 3.2.10 (Process renaming) *When* $\mathcal{O} \mid \Gamma_0, \Delta_0 \vdash M^+ : \varphi^+$ *is derivable,*

$$\mathcal{O} \mid \Gamma_0^m, \Delta_0^m \vdash \{m \mapsto M'\} : \varphi^{+m}$$

is also derivable for some local term M'.

Proof By induction on the height of derivation. All rules except com and EC are trivial because they do not interact with the modalities. For the EC rule, we have to apply the induction hypothesis twice to change modalities in two components. For the com rule, let us assume the derivation ends in the com rule as:

$$\frac{\mathcal{O}_0 \mid \Gamma \vdash M : [i]\varphi \qquad \mathcal{O}_1 \mid \Delta \vdash N : [j]\psi}{\mathcal{O}_0 \mid \mathcal{O}_1 \mid \Gamma \vdash (*^{\rightarrow d}_{\leftarrow c})M : [i]\psi \mid \Delta \vdash (*^{\rightarrow c}_{\leftarrow d})N : [j]\varphi}\ .$$

We have to consider three cases: first, when $\Gamma_0, \Delta_0 \vdash M^+ : \varphi^+$ in the statement is in $[\mathcal{O}_0, \mathcal{O}_1]$; second, when $\Gamma_0, \Delta_0 \vdash M^+ : \varphi^+$ in the statement is identical to $\Gamma \vdash (*^{\rightarrow d}_{\leftarrow c})M : [i]\psi$; and third, when $\Gamma_0, \Delta_0 \vdash M^+ : \varphi^+$ in the statement is identical to $\Delta \vdash (*^{\rightarrow c}_{\leftarrow d})N : [j]\varphi$. The second and third cases are symmetric. In the first case, M^+ is actually a concurrent construct $([P_i, Q_i])_{i \in I}$. We can apply the induction hypothesis to P and Q and combine them again $\{m \mapsto [P', Q']\}$. In the second case, we can use the induction hypothesis on the left branch[3]. ■

Strong Normalization

For proving strong normalization, we use an auxiliary relation, which is similar to the reduction $\rightsquigarrow$. We denote the relation by $\leadsto$. The $\leadsto$ relation (symmetric reduction) is also a congruence. All reductions are symmetric reductions. In addition to reductions, symmetric reductions contain pairs like $[M, N] \leadsto N$ when N does not contain any abort, $*^{\rightarrow d}_{\leftarrow c}$ or $*_{\leftarrow c}$. Ordinary reductions are not symmetric because it allows $(S, \{i \mapsto [M, N]\}) \rightsquigarrow_{\mathrm{A}} (S, \{i \mapsto M\})$ but not $(S, \{i \mapsto [M, N]\}) \rightsquigarrow_{\mathrm{A}} (S, \{i \mapsto N\})$. This asymmetry allows us to encode waitfree computation in our calculus. However, it is easier to prove strong normalization for the symmetric reductions. When strong normalization for the symmetric reductions holds, the same property holds for the reductions because there are less pairs in reductions than in symmetric reductions.

[3]One crucial thing is our choice of the form of the com rule. If we use the com' rule by Avron [8], the proof does not proceed because the contexts Γ and Δ are duplicated as

$$\frac{\mathcal{O}_0 \mid \Gamma, \Delta \vdash M : [i]\varphi \qquad \mathcal{O}_1 \mid \Gamma, \Delta \vdash N : [j]\psi}{\mathcal{O}_0 \mid \mathcal{O}_1 \mid \Gamma \vdash (*^{\rightarrow d}_{\leftarrow c})M : [i]\psi \mid \Delta \vdash (*^{\rightarrow c}_{\leftarrow d})N : [j]\varphi}\ .$$

There, if we want to change the modalities in the rightmost component naively, we also have to change the modalities in the second rightmost component.

Theorem 3.2.11 (Strong normalization of $\boldsymbol{\lambda}$-GD) *λ-GD is strongly normalizing.*

Proof For proving this, we consider the *pure fragment* that does not contain $(*\overset{\to d}{\underset{\leftarrow c}{}})M$, $(*\overset{\to c}{\underset{\leftarrow d}{}})N$. We first reduce the strong normalization of the λ-GD to that of the pure fragment, and ultimately to that of de Groote's natural deduction with permutation-conversion [41][4].

We assume an infinitely long sequence of reductions, namely, $(S_0, \mathcal{O}_0) \rightsquigarrow (S_1, \mathcal{O}_1) \rightsquigarrow (S_2, \mathcal{O}_2) \rightsquigarrow \cdots$. From this, we are going to construct an infinitely long sequence of symmetric reductions in the pure fragment.

For that, we first build an infinite reduction sequence with constant stores. Using the original infinite sequence, we define a store called the *store prophecy* S_∞ where $S_\infty(c)$ is defined to be ϵ if $S_k(c) = \epsilon$ holds for all $k \in \omega$ and $S_\infty(c)$ is defined to be M if $S_k(c) = M$ holds for some $k \in \omega$. Since store contents are never overwritten, S_∞ is well defined. Moreover, $S_i(c)$ and $S_\infty(c)$ coincide unless $S_i(c) = \epsilon$.

We build another reduction sequence $(S_\infty, \mathcal{O}_0) \rightsquigarrow^* (S_\infty, \mathcal{O}'_1) \rightsquigarrow^* (S_\infty, \mathcal{O}'_2) \rightsquigarrow^* \cdots$ with the following invariant: $\mathcal{O}'_i$ can be obtained by replacing some abort occurrences in $\mathcal{O}_i$ with some terms. More specifically, we translate each reduction as follows, keeping the invariant inductively on the number of steps (the base case is satisfied by $\mathcal{O}'_0 = \mathcal{O}_0$ immediately):

- a read reduction $(S_k, \mathcal{C}[*_{\leftarrow c}]) \rightsquigarrow_{\mathrm{R}} (S_{k+1}, \mathcal{C}[O])$ is translated into $(S_\infty, \mathcal{C}'[*_{\leftarrow c}]) \rightsquigarrow_{\mathrm{R}} (S_{k+1}, \mathcal{C}'[O'])$ for a context $\mathcal{C}'$. If $S_i(c)$ is a term, $S_\infty(c)$ and O' are also identical to the term. Otherwise, O' must be abort. Thus, the invariant holds for $k+1$.

- a write reduction $(S[c \mapsto \epsilon], \mathcal{C}[(*\overset{\to c}{\underset{\leftarrow d}{}})M]) \rightsquigarrow_{\mathrm{W}} (S[c \mapsto M], \mathcal{C}[*_{\leftarrow d}])$ is translated into $(S_\infty, \mathcal{C}'[(*\overset{\to c}{\underset{\leftarrow d}{}})M]) \rightsquigarrow_{\mathrm{W}} (S_\infty, \mathcal{C}'[*_{\leftarrow d}])$;

- an abort reduction of the form $(S_k, \mathcal{C}[C[\mathsf{abort}]]) \rightsquigarrow_{\mathrm{A}} (S_{k+1}, \mathcal{C}[\mathsf{abort}])$ can be translated either to a similar reduction or no reduction if the abort in the redex is replaced by another term in the $\mathcal{O}'_k$. Note that even in that case, the result $\mathcal{O}'_{k+1}$ can be obtained by replacing some abort occurrences in $\mathcal{O}_{k+1}$ with other terms;

- any other reduction $(S_k, \mathcal{C}[M]) \rightsquigarrow_{\mathrm{B/P}} (S_{k+1}, \mathcal{C}[N])$ is translated into one similar reduction $(S_\infty, \mathcal{C}'[M']) \rightsquigarrow_{\mathrm{B/P}} (S_\infty, \mathcal{C}'[N'])$.

[4]To the same effect, we might be able to use other strong normalization results for lambda calculi with commutative conversions, like Balat, Di Cosmo and Fiore [15].

Here, we have to show that the translated sequence of symmetric reductions is infinite. For that, we can use the facts that there are only finite number of mentioned locations and that each location allows only one write, and that an abort propagation always strictly shortens the term under operation.

After that, we can replace every $*_{\leftarrow c}$ with $S_\infty(c)$. Since $*_{\leftarrow c}$ either reduces to $S_\infty(c)$ or abort, replacing it with $S_\infty(c)$ will only "shorten" the reduction sequence for at most one read step. Replacing all readers makes an infinite reduction sequence in the pure fragment. Moreover, the result of the translation is also well typed[5]. A typing derivation of the resulting hyperterm can be obtained by replacing com rules with EW rules and changing the process number in global types of some variables (c.f. the proof of Theorem 3.2.9).

We are aiming at reducing the problem to the strong normalization result by de Groote [41]. Since we have eliminated $(*^{\rightarrow d}_{\leftarrow c})M$ and $(*^{\rightarrow c}_{\leftarrow d})N$ occurrences, the remaining difference from de Groote's lambda calculus [41] is small: some abort propagation reductions and some permutative reductions involving $[M, N]$. We just have to make sure that there are no infinite reduction sequences that consist of only these two kinds of reductions. We can deal with the permutative reductions following de Groote [41]'s strategy for introducing $\perp$. There are no infinite sequence of abort propagation reductions keeping the number of $[M, N]$ constructions; and an abort propagation reduction cannot increase the number of $[M, N]$ constructions in a configuration. Combined, there are no infinite sequence of abort propagation reductions. ■

Strong normalization and non-abortfullness hold even for reductions where $*^{\rightarrow d}_{\leftarrow c}$ operators are copied or discarded. This is different from the situation in Chapter 2 where discarding term c causes the peer $\bar{c}$ to deadlock. This is the reason why we can use intuitionistic propositional logic rather than linear (or affine) logics in this chapter.

3.3 Typed Waitfreedom

Waitfree protocols [70, 125] are a class of protocols that can solve some of the input-output problems [19, 110]. If a waitfree protocol solves an input-output problem, then we call the protocol to be in the *waitfreedom*. We define the typed version of waitfreedom. Since these definitions are involved, we supply many examples.

[5]Because each channel c is uniquely associated with a type φ in the original derivation and $S_\infty(c)$ is a closed term of type φ.

3.3.1 Typed Input-Output Problem

Saks and Zaharoglou [125] formulated waitfreedom as a class of input-output problems. Given inputs for all processes and outputs of all processes, an input-output problem decides whether the processes have succeeded or not. We change the standard definition and have typed terms as inputs and outputs. This change is necessary because according to the original definition of waitfreedom, a single process waitfree protocol can solve any undecidable problem because a waitfree protocol can contain arbitrary functions.

In order to define the set of inputs and outputs, we let $\mathcal{T}^{-}(\varphi)$ denote the set of closed, pure terms of type φ, and $\mathcal{V}^{-}(\varphi)$ denote the set of normal terms in $\mathcal{T}^{-}(\varphi)$. For a finite set of processes $\mathbb{P}$, a *typed input-output problem* consists of each process's input type $(\iota_i)_{i\in\mathbb{P}}$, each process's output type $(o_i)_{i\in\mathbb{P}}$, and a task $R \subseteq \prod_{i\in\mathbb{P}} (\mathcal{T}^{-}(\iota_i)) \times \prod_{i\in\mathbb{P}} (\mathcal{V}^{-}(o_i))$. In words, a task R decides whether a pair of inputs and outputs is good or not.

Example 3.3.1 (The addition problem) *Let the set $\mathbb{P}$ be $\{0,1\}$, the types ι_0, ι_1, o_0 and o_1 be natural numbers*[6]*. Now we can define a task requiring at least one of the processes outputs the addition of the inputs. Specifically,*

$$\begin{aligned} R_{+} =& \{\langle\{0 \mapsto x, 1 \mapsto y\}, \{0 \mapsto (x+y), 1 \mapsto z\}\rangle \mid x,y,z \in \mathbb{N}\} \\ & \cup \{\langle\{0 \mapsto x, 1 \mapsto y\}, \{0 \mapsto z, 1 \mapsto (x+y)\}\rangle \mid x,y,z \in \mathbb{N}\} \ . \end{aligned}$$

The equation above says the task R_{+} allows two different kinds of behavior. The first clause says if process 0 receives x and process 1 receives y as their inputs, and process 0 outputs $x+y$; then whatever process 1 outputs, the two processes are considered to have solved the task. The second clause poses a symmetric condition where the two processes are swapped. In order to solve the task R_{+}, it is enough for one of the two processes to give the correct answer.

Example 3.3.2 (The exchange problem) *There is another problem that requires two processes to exchange their inputs. Namely,*

$$R_{\texttt{exch}} = \{\langle\{0 \mapsto x, 1 \mapsto y\}, \{0 \mapsto y, 1 \mapsto x\}\rangle \mid x,y \in \{\textit{tt}, \textit{ff}\}\} \ .$$

[6]Actually, the type of natural numbers is not definable in our language. However, since we can define a three-valued type, we can make the same argument below using $\mathbb{Z}/3$ instead of natural numbers.

3.3.2 Typed Protocols

We assume a finite set $\mathbb{P}$ of processes and a countably infinite set of *program variables* $\mathsf{ProV} = \{\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots\}$. Program variables are typeset in the typewriter font. We assume an injection from variables to program variables $x \mapsto \mathtt{x}_x$, whose image leaves infinitely many unused program variables.

A *program* is defined by BNF:

$$p ::= \epsilon \mid \mathtt{x} \leftarrow E; p \mid c \leftarrow E; p$$

where c is a location and an *expression* E is

$$E ::= x \mid \mathtt{x} \mid c \mid (EE) \mid \lambda x.E \mid \langle E, E\rangle \mid \mathsf{inl}\,(E) \mid \mathsf{inr}\,(E) \mid$$
$$\pi_{\mathsf{l}}\,(E) \mid \pi_{\mathsf{r}}\,(E) \mid \mathsf{match}\, E\, \mathsf{of}\; \mathsf{inl}(x).E/\mathsf{inr}(y).E \mid \epsilon \ .$$

Here, the expression ϵ is used as the initial placeholder of the shared memory.

A program is *well formed* when any program variable $\mathtt{x}$ (resp. location c) first appears in a $\mathtt{x} \leftarrow E$ (resp. $c \leftarrow E$) sentence where E does not contain $\mathtt{x}$ (resp. c), and after that, only appears in expressions. In other words, a well-formed program is in the single assignment form.

A *contexted type* $\Gamma \vdash \varphi^{+}$ is a sequent without a term but with variables in Γ. For a contexted type $(\Gamma \vdash \varphi^{+})$, we write $M\colon(\Gamma \vdash \varphi^{+})$ for $\Gamma \vdash M\colon\varphi^{+}$. For input types $(\iota_i)_{i\in\mathbb{P}}$ and output types $(o_i)_{i\in\mathbb{P}}$, a *typed protocol*[7] has:

- two program variables $\mathtt{i}_i$ and $\mathtt{o}_i$ for each process i;
- a finite set of locations C;
- two functions $w\colon C \to \mathbb{P}$ and $r\colon C \to \mathbb{P}$ (specifying the writer and the reader of each location);
- W: maps a location in C to a contexted type;
- V: a finite set of program variables that contains $\mathtt{i}_i$'s and $\mathtt{o}_i$'s;
- a function t_i for each $i \in \mathbb{P}$ that maps a program variable in V to a contexted type $(x_k\colon[i]\varphi_k)_k \vdash [i]\varphi$ with a special condition $t_i(\mathtt{i}_i) = \iota_i$;
- a typed program p_i for each $i \in \mathbb{P}$, where a *typed program* is a well-formed program where all sentences are typed according to the rules below. A sentence

[7]We formulated this definition as close as we can to the definition of waitfree protocols in Saks and Zaharoglou [126, **2.3.**].

$\mathtt{x} \leftarrow E$ is typed iff $\vdash E\colon [i]t_i(\mathtt{x})$ is derivable with assumptions of the form $\vdash \mathtt{y}\colon [i]t_i(\mathtt{y})$ and $\vdash c\colon W(c)$. A sentence $c \leftarrow E$ is typed iff $\vdash E\colon [i]W(c)$ is derivable with assumptions of the form $\vdash \mathtt{y}\colon [i]t(\mathtt{y})$ and $\vdash c\colon [i]W(c)$.

Example 3.3.3 (The addition protocol) *We give a concrete typed protocol that solves the addition task (Example 3.3.1). The set C of locations contains two elements c and d. The locations c and d will be used as follows: process 0 writes to d and reads from c, and process 1 writes to c and reads from d. Formally, $w(c) = 1$, $w(d) = 0$, $r(c) = 0$ and $r(d) = 1$. Both locations will contain natural numbers. Formally, $W(c) = W(d) = \mathbb{N}$. The set of program variables V is just $\{\boldsymbol{i}_0, \boldsymbol{i}_1, \boldsymbol{o}_0, \boldsymbol{o}_1\}$. The typed programs are*

$$p_0 = d \leftarrow \boldsymbol{i}_0; \boldsymbol{o}_0 \leftarrow \mathit{add}(c, \boldsymbol{i}_0); \boldsymbol{o}_0 \leftarrow \boldsymbol{i}_0 \text{ and}$$
$$p_1 = c \leftarrow \boldsymbol{i}_1; \boldsymbol{o}_1 \leftarrow \mathit{add}(d, \boldsymbol{i}_1); \boldsymbol{o}_1 \leftarrow \boldsymbol{i}_1$$

where $\mathit{add}(M, N)$ is a specially introduced[8] term that reduce to the sum. The last two sentences of each program write to the same variable. Although a variable is never updated after being filled, the second write is still meaningful because the first write may fail. We omit the type assignments t_0, t_1 and typing derivations.

3.3.3 Typed Waitfree Computation

We define when a typed protocol solves a typed input-output problem. This definition is obtained by modifying Saks and Zaharoglou's definition of input-output problems [125].

A closed local term M is *of* type φ iff there is a derivation of $\vdash \{0 \mapsto M\}\colon [0]\varphi$. Let $\mathbb{P}$ be $\{0, \ldots, n-1\}$ and fix a typed protocol. We are going to define a virtual machine for executing the typed protocol. In order to do that, we first define snapshots of the virtual machine. After this, we will define the transition from snapshots to snapshots.

Definition 3.3.4 *A program variable content for $i \in \mathbb{P}$ is a partial function that maps a program variable to a closed local term of $t_i(\boldsymbol{x})$. A global term M^+ is of a contexted type $\Gamma \vdash \varphi^+$ when $\Gamma \vdash M^+\colon\varphi^+$ is derivable. A process snapshot of $i \in \mathbb{P}$ is a tuple $\langle p, m\rangle$ where p is either a program or* abort *and m is a program variable content for i. We let S_i denote the set of process snapshots for i. A system snapshot is a pair $\langle \overrightarrow{s}, \overrightarrow{v}\rangle$ of process snapshots and a shared memory snapshot, where $\overrightarrow{s} = \langle s_0, s_1, \ldots, s_{n-1}\rangle \in \prod_{i\in\mathbb{P}}(S_i)$ and $\overrightarrow{v} = (v_c)_{c\in C} \in \prod_{c\in C}(\mathcal{V}(W(c)) \cup \{\epsilon\})$.*

[8]If we choose $\mathbb{Z}/3$, we do not need to introduce a special term, but we can define addition using nested match constructs.

Definition 3.3.5 *For a nonempty subset J of $\mathbb{P}$, we define an operator $\lhd J$ that takes a system snapshot and produces a system snapshot. This operator depicts a computational step where the processes in J are fired.*

We define $(\overrightarrow{s}, \overrightarrow{v}) \lhd J = (\overrightarrow{u}, \overrightarrow{m})$ by defining u_i and m_c for $i \in \mathbb{P}$ and $c \in C$. Let s_i be $\langle p, x\rangle$: when i is not in J, u_i is identical to s_i; otherwise, when i is in J:

$$u_i = \begin{cases} \langle p', x\rangle & (\textit{if } p = c \leftarrow E; p' \textit{ and } [\![E]\!]_{x,\overrightarrow{v}} = \epsilon) \\ \langle p', x[\boldsymbol{x} \mapsto [\![E]\!]_{x,\overrightarrow{v}}]\rangle & (\textit{if } p = \boldsymbol{x} \leftarrow E; p', \quad x(\boldsymbol{x}) = \epsilon \textit{ and } [\![E]\!]_{x,\overrightarrow{v}} \neq \epsilon) \\ \langle p', x\rangle & (\textit{if } p = \boldsymbol{x} \leftarrow E; p' \textit{ but } x(\boldsymbol{x}) \neq \epsilon \textit{ or } [\![E]\!]_{x,\overrightarrow{v}} = \epsilon) \\ \langle \epsilon, x\rangle & (\textit{if } p = \epsilon) \end{cases}$$

$$m_c = \begin{cases} [\![E]\!]_{x,\overrightarrow{v}} & (\textit{if } p = c \leftarrow E; p', \quad w(c) = i \textit{ and } v_c = \epsilon) \\ v_c & (\textit{otherwise}) \end{cases}$$

with the following notations. The term $[\![E]\!]_{x,\overrightarrow{v}}$ is defined as the unique normal form of the local term $E[x(\overrightarrow{\boldsymbol{y}})/\overrightarrow{\boldsymbol{y}}][\overrightarrow{v_c}/\overrightarrow{c}]$, where every program variable $\boldsymbol{y}$ is replaced by $x(\boldsymbol{y})$ and every location c is replaced by v_c. If any of the substitutes is ϵ, $[\![E]\!]_{x,\overrightarrow{v}}$ is defined to be ϵ.

Definition 3.3.6 *A block is a nonempty subset of $\mathbb{P}$. A schedule is an infinite sequence of blocks, which looks like $\sigma = \sigma_0\sigma_1\sigma_2\cdots$. We say i is non-faulty (resp. faulty) in σ if it appears infinitely (resp. only finitely many) often. When every process is non-faulty, the schedule is fair.*

Example 3.3.7 (An example of schedules) *When $\mathbb{P}$ is a singleton $\{0\}$, there is only one schedule $(\{0\}, \{0\}, \{0\}, \ldots)$ where "$\ldots$" contains only $\{0\}$'s. Since 0 is the only process and 0 is non-faulty, this schedule is fair.*

Assume $\mathbb{P} = \{0, 1\}$. There is a schedule which is not fair: $(\{0\}, \{0\}, \{0\}, \ldots)$ where "$\ldots$" contains only $\{0\}$'s. Even when we add a finite prefix to an unfair schedule, the schedule is still not fair because no process can appear infinitely often in that finite prefix.

Obviously, when a schedule contains infinitely many $\{0, 1\}$'s, then the schedule is fair. Not only that, but when a schedule contains infinitely many $\{0\}$'s and infinitely many $\{1\}$'s, then the schedule is fair. In the other direction, a fair schedule must satisfy one of these conditions.

Definition 3.3.8 *A run is a triple $\langle \Pi, \overrightarrow{x}, \sigma \rangle$, where Π is a typed protocol, $\overrightarrow{x} \in \prod_{i \in \mathbb{P}} \mathcal{T}^{-}(\iota_i)$ is the input, and σ is a schedule. The execution associated to the run is defined as the infinite sequence of system snapshots $C_0 C_1 C_2 \cdots$, where $C_0 = \langle \overrightarrow{s^0}, \overrightarrow{v^0} \rangle$ is defined by $\overrightarrow{s^0_i} = \langle p_i, [\mathit{i}_i \mapsto x_i] \rangle$ and $v_c = \epsilon$, and C_{k+1} is defined to be $C_k \lhd \sigma_k$. Since all programs are finite, the system snapshot will stay constant after initial finite steps. We call that constant system snapshot the final system snapshot of an execution.*

Example 3.3.9 (An example of a run and an execution) *Let $\mathbb{P}$ be $\{0,1\}$ and Π be the typed protocol given in Example 3.3.3. let us take $x_0 = 3$ and $x_1 = 4$. Also we choose a schedule σ as $\{0\}, \{0\}, \{0,1\}, \{1\}, \{1\}$, followed by an infinite sequence of $\{0,1\}$'s. The triple $\langle \Pi, \overrightarrow{x}, \sigma \rangle$ constitutes a run.*

We can associate an execution with this run. The initial system snapshot $C_0 = \langle \overrightarrow{s^0}, \overrightarrow{v^0} \rangle$ contains process snapshots

$$s^0_0 = \langle p_0, \{\mathit{i}_0 \mapsto 3\} \rangle \text{ and } s^0_1 = \langle p_1, \{\mathit{i}_1 \mapsto 4\} \rangle$$

where p_0 and p_1 are defined in Example 3.3.3. In the system snapshot, $\overrightarrow{v^0}$ maps both locations c and d to ϵ. The next system snapshot C_1 is defined to be $C_0 \lhd \sigma_0$. Since $\sigma_0 = \{0\}$, it does not contain 1. So, s^1_1 is the same as s^0_1. On the other hand, since p_0 is $d \leftarrow \mathit{i}_0; \mathit{o}_0 \leftarrow add(c, \mathit{i}_0); \mathit{o}_0 \leftarrow \mathit{i}_0$, s^1_0 is defined to be $\langle \mathit{o}_0 \leftarrow add(c, \mathit{i}_0); \mathit{o}_0 \leftarrow \mathit{i}_0, \{\mathit{i}_0 \mapsto 3\} \rangle$ and v^1_d is defined to be 3. However, v^1_c is the same as v^0_c and thus remains to be ϵ. We conclude that $C_1 = \langle \{0 \mapsto s^1_0, 1 \mapsto s^1_1\}, \{c \mapsto \epsilon, d \mapsto 3\} \rangle$. Again, the next snapshot C_2 is defined to be $C_1 \lhd \sigma_1$. Since $\sigma_1 = \{0\}$, process 1's state is not changed so that $s^2_0 = s^1_0 = \langle p_1, \{\mathit{i}_1 \mapsto 4\} \rangle$. On the other hand, since $s^1_0 = \langle \mathit{o}_0 \leftarrow add(c, \mathit{i}_0); \mathit{o}_0 \leftarrow \mathit{i}_0, \{\mathit{i}_0 \mapsto 3\} \rangle$ and 0 is in σ_1, process 0's state is updated to $s^2_0 = \langle \mathit{o}_0 \leftarrow \mathit{i}_0, \{\mathit{i}_0 \mapsto 3\} \rangle$ where the process failed to read from the location c because v^1_c is ϵ. As a result, the program variable o_i obtained no assignments. The store contents are not updated either so that $v^2_c = v^1_c = \epsilon$ and $v^2_d = v^1_d = 3$. After four more rounds (including $i = 6$ to confirm $C_5 = C_6$), we obtain an execution shown in Table 3.1. Since $C_i = C_5$ for $i > 5$, the system snapshot C_5 is the final system snapshot of this execution.

Definition 3.3.10 *Process i's output $\hat{o_k}$ at step k is M if the i-th process snapshot of C_k is (p, x) and the $x[\mathit{o}_i] = M$, which might be ϵ. The decision value of i on the run $\langle \Pi, \overrightarrow{x}, \sigma \rangle$, denoted $d_i \in \mathcal{V}^{-}(o_i) \cup \{\epsilon\}$ is the first non-ϵ output in the sequence $(\hat{o_k})_{k \in \omega}$, or ϵ if such an output does not exist. The decision vector of the run is the n-tuple $\overrightarrow{d}$ consists of decision value d_i's.*

Table 3.1: An example of execution of a typed protocol defined in Example 3.3.3.

i	s_0^i	s_1^i	v_c^i	v_d^i	σ_i
0	$\langle d \leftarrow \mathtt{i}_0; \mathtt{o}_0 \leftarrow add(c, \mathtt{i}_0); \mathtt{o}_0 \leftarrow \mathtt{i}_0, \{\mathtt{i}_0 \mapsto 3\}\rangle$	$\langle c \leftarrow \mathtt{i}_1; \mathtt{o}_1 \leftarrow add(d, \mathtt{i}_1); \mathtt{o}_1 \leftarrow \mathtt{i}_1, \{\mathtt{i}_1 \mapsto 4\}\rangle$	ϵ	ϵ	$\{0\}$
1	$\langle \mathtt{o}_0 \leftarrow add(c, \mathtt{i}_0); \mathtt{o}_0 \leftarrow \mathtt{i}_0, \{\mathtt{i}_0 \mapsto 3\}\rangle$	$\langle c \leftarrow \mathtt{i}_1; \mathtt{o}_1 \leftarrow add(d, \mathtt{i}_1); \mathtt{o}_1 \leftarrow \mathtt{i}_1, \{\mathtt{i}_1 \mapsto 4\}\rangle$	ϵ	3	$\{0\}$
2	$\langle \mathtt{o}_0 \leftarrow \mathtt{i}_0, \{\mathtt{i}_0 \mapsto 3\}\rangle$	$\langle c \leftarrow \mathtt{i}_1; \mathtt{o}_1 \leftarrow add(d, \mathtt{i}_1); \mathtt{o}_1 \leftarrow \mathtt{i}_1, \{\mathtt{i}_1 \mapsto 4\}\rangle$	ϵ	3	$\{0{,}1\}$
3	$\langle \epsilon, \{\mathtt{i}_0 \mapsto 3, \mathtt{o}_0 \mapsto 3\}\rangle$	$\langle \mathtt{o}_1 \leftarrow add(d, \mathtt{i}_1); \mathtt{o}_1 \leftarrow \mathtt{i}_1, \{\mathtt{i}_1 \mapsto 4\}\rangle$	4	3	$\{1\}$
4	$\langle \epsilon, \{\mathtt{i}_0 \mapsto 3, \mathtt{o}_0 \mapsto 3\}\rangle$	$\langle \mathtt{o}_1 \leftarrow \mathtt{i}_1, \{\mathtt{i}_1 \mapsto 4, \mathtt{o}_1 \mapsto 7\}\rangle$	4	3	$\{1\}$
≥ 5	$\langle \epsilon, \{\mathtt{i}_0 \mapsto 3, \mathtt{o}_0 \mapsto 3\}\rangle$	$\langle \epsilon, \{\mathtt{i}_1 \mapsto 4, \mathtt{o}_1 \mapsto 7\}\rangle$	4	3	$\{0, 1\}$

A vector $\overrightarrow{b} \in \prod_{i\in\mathbb{P}}(\mathcal{V}^{-}(o_i))$ is *compatible with* $\overrightarrow{d} \in \prod_{i\in\mathbb{P}}(\mathcal{V}^{-}(o_i) \cup \{\epsilon\})$ iff $d_i = b_i$ or $d_i = \epsilon$ holds for any process i. For a task R, an input $\overrightarrow{x} \in \prod_{i\in\mathbb{P}}\mathcal{T}^{-}(\iota_i)$ is *R-permissible* iff there is at least one vector $\overrightarrow{b} \in \prod_{i\in\mathbb{P}}(\mathcal{V}^{-}(o_i))$ with $(\overrightarrow{x}, \overrightarrow{b}) \in R$.

Definition 3.3.11 *A typed protocol Π solves the typed input-output problem $\langle(\iota_i)_{i\in\mathbb{P}}, (o_i)_{i\in\mathbb{P}}, R\rangle$ on schedule σ iff for all R-permissible inputs $\overrightarrow{x}$ and a schedule σ, the decision value of every non-faulty process i is a term M not ϵ, and there is a vector $\overrightarrow{b} \in \prod_{i\in\mathbb{P}}(\mathcal{V}^{-}(o_i))$ with $\langle\overrightarrow{x}, \overrightarrow{b}\rangle \in R$ which is compatible with the decision vector $\overrightarrow{d}$ of the run $\langle\Pi, \overrightarrow{x}, \sigma\rangle$. A typed protocol is waitfree iff it solves the problem on every schedule σ. In that case, the typed input-output problem is waitfreely solvable.*

Example 3.3.12 (Permissible inputs and outputs) *The compatible inputs for the addition problem in Example 3.3.1 are pairs of natural numbers. For the input $(3, 4)$, the addition of the inputs is 7, but the addition problem requires at least one process to output the sum. Moreover, when process 0 is faulty, process 0 is allowed to output ϵ. In that case, whatever natural number process 1 outputs, the addition problem is solved because any $(\epsilon, n)(n \in \mathbb{N})$ is compatible with some vector $\overrightarrow{b}$ in R_+. In short, in the addition problem, if one process outputs ϵ, the other process is allowed to output whatever.*

Example 3.3.13 (An example of a typed protocol solving the addition task) *In the execution in Table 3.1, we can replace 3 and 4 with any natural numbers and 7 with the sum. This shows that the typed protocol in Example 3.3.3 solves the addition problem in Example 3.3.1 on the schedule*

$$\sigma = (\{0\}, \{0\}, \{0, 1\}, \{1\}, \{1\}, \{0, 1\}, \{0, 1\}, \ldots)$$

where the last "$\ldots$" represents an infinite sequence consisting of $\{0, 1\}$'s. On the schedule σ, both processes 0 and 1 are non-faulty. In Table 3.1, the decision value of process 0 is $d_0 = 3$ and the decision value of process 1 is $d_1 = 7$. The tuple $\langle(x_0, x_1), (d_0, d_1)\rangle$ is in R_+. The same argument goes for different (x_0, x_1)'s. Thus, the typed protocol solves the addition problem on schedule σ. Actually, the typed protocol solves the problem for all schedules. To see why, we can do case analysis on an arbitrarily taken schedule: one case where all processes produce non-ϵ outputs and the other case where one of the prcesses produces a non-ϵ output. There is no possibility where no processes produce non-ϵ outputs because there are infinitely many blocks in a schedule and each block contains at least one process (by definition of a block). In the first case, when the first block contains 0, the execution proceeds like in Table 3.1.

Otherwise, the first block must contain 1 so that the execution proceeds like Table 3.1 with 0 and 1 swapped. In either case, the typed protocol solves the task. In the second case, the output producing process can produce whatever output to meet the task because the other process produces ϵ.

Example 3.3.14 (A waitfreely unsolvable typed input-output problem) *On the other hand, there is no typed protocol that solves the exchange problem (Example 3.3.2). To see the reason, take any typed protocol. There exist natural numbers m and n so that the typed protocol contains m commands for process 0 and n commands for process 1. Consider a schedule that consists of the first m blocks $\{0\}$ and the following n blocks $\{1\}$ and the rest $\{0, 1\}$. Thus process 0 finishes before process 1 starts so that process 0 cannot learn process 1's input. Because of the infinitely many $\{0, 1\}$'s at the tail, under the schedule, both processes are non-faulty so that they must produce the correct outputs. However, since process 0 finishes before process 1 starts, there is no way process 0 can output process 1's input, not by chance. Even when process 0 produces the correct output by chance, when we change process 1's input, then, process 0's output is wrong with regard to process 1's new input.*

3.4 Characterization of Waitfreedom by λ-GD

In this section, we show that the ability of the waitfree protocols and λ-GD are the same.

Definition 3.4.1 *A typed input-output problem $\langle (\iota_i)_{i\in\mathbb{P}}, (o_i)_{i\in\mathbb{P}}, R\rangle$ is solvable by a global term M^+ of contexted type $(x_i : [i]\iota_i)_{i\in\mathbb{P}} \vdash (o_i)_{i\in\mathbb{P}}$ iff for any closed $(N_i)_{i\in\mathbb{P}}$ of ι_i, all normal forms of $M^+[\overrightarrow{N_i}/\overrightarrow{x_i}]$ are in the form $(V_i)_{i\in\mathbb{P}}$ where $\langle (N_i)_{i\in\mathbb{P}}, (V_i)_{i\in\mathbb{P}}\rangle \in R$.*

Example 3.4.2 (A global term solving the addition problem) *For the exclusive-or problem (defined in Example 3.3.1), there is a global term $\{0 \mapsto [add((*\overset{\rightarrow d}{\underset{\leftarrow c}{}})x_0, x_0), x_0], 1 \mapsto [add((*\overset{\rightarrow c}{\underset{\leftarrow d}{}})x_1, x_1), x_1]\}$ that solves the problem. The global term can be typed as in Figure 3.6.*

Now we can state the most important results in this chapter.

Theorem 3.4.3 (Soundness of λ-GD with regard to waitfreedom) *If a typed input-output problem is solvable by a global term, there exists a typed protocol that solves the problem.*

In order to prove this theorem, we are going to translate a typed hyperterm into a typed protocol inductively on the type derivations. To make induction work, we

Part A

$$\dfrac{\dfrac{\dfrac{}{x\colon[0]\mathbb{N}\vdash x\colon[0]\mathbb{N}}\,\mathrm{Ax}\qquad \vdots}{x\colon[0]\mathbb{N}\vdash\{0\mapsto add((*^{\to d}_{\leftarrow c})x,x)\}\colon[0]\mathbb{N}\ \rule[-0.15em]{0.25em}{0.9em}\ y\colon[1]\mathbb{N}\vdash\{1\mapsto(*^{\to c}_{\leftarrow d})y\}\colon[1]\mathbb{N}}\,add\qquad \dfrac{}{y\colon[1]\mathbb{N}\vdash y\colon[1]\mathbb{N}}\,\mathrm{Ax}}{x\colon[0]\mathbb{N}\vdash\{0\mapsto add((*^{\to d}_{\leftarrow c})x,x)\}\colon[0]\mathbb{N}\ \rule[-0.15em]{0.25em}{0.9em}\ y\colon[1]\mathbb{N}\vdash\{1\mapsto add((*^{\to c}_{\leftarrow d})y,y)\}\colon[1]\mathbb{N}}\,add$$

Part B

$$\dfrac{\dfrac{\dfrac{}{y\colon[1]\mathbb{N}\vdash y\colon[1]\mathbb{N}}\,\mathrm{Ax}\qquad\vdots}{x\colon[0]\mathbb{N}\vdash\{0\mapsto add((*^{\to d}_{\leftarrow c})x,x),1\mapsto y\}\colon\{0\mapsto\mathbb{N},1\mapsto\mathbb{N}\}\ \rule[-0.15em]{0.25em}{0.9em}\ y\colon[1]\mathbb{N}\vdash\{1\mapsto add((*^{\to c}_{\leftarrow d})y,y)\}\colon[1]\mathbb{N}}\,\wedge\mathcal{I}\qquad\dfrac{}{x\colon[0]\mathbb{N}\vdash x\colon[0]\mathbb{N}}\,\mathrm{Ax}}{x\colon[0]\mathbb{N},y\colon[1]\mathbb{N}\vdash\{0\mapsto add((*^{\to d}_{\leftarrow c})x,x),1\mapsto y\}\colon\{0\mapsto\mathbb{N},1\mapsto\mathbb{N}\}\ \rule[-0.15em]{0.25em}{0.9em}\ x\colon[0]\mathbb{N},y\colon[1]\mathbb{N}\vdash\{0\mapsto x,1\mapsto add((*^{\to c}_{\leftarrow d})y,y)\}\colon\{0\mapsto\mathbb{N},1\mapsto\mathbb{N}\}}\,\wedge\mathcal{I}$$

Whole Derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x\colon[0]\mathbb{N}\vdash\{0\mapsto x\}\colon[0]\mathbb{N}}\,\mathrm{Ax}\qquad\dfrac{}{y\colon[1]\mathbb{N}\vdash\{1\mapsto y\}\colon[1]\mathbb{N}}\,\mathrm{Ax}}{x\colon[0]\mathbb{N}\vdash\{0\mapsto(*^{\to d}_{\leftarrow c})x\}\colon[0]\mathbb{N}\ \rule[-0.15em]{0.25em}{0.9em}\ y\colon[1]\mathbb{N}\vdash\{1\mapsto(*^{\to c}_{\leftarrow d})y\}\colon[1]\mathbb{N}}\,\text{01-com}}{x\colon[0]\mathbb{N}\vdash\{0\mapsto add((*^{\to d}_{\leftarrow c})x,x)\}\colon[0]\mathbb{N}\ \rule[-0.15em]{0.25em}{0.9em}\ y\colon[1]\mathbb{N}\vdash\{1\mapsto add((*^{\to c}_{\leftarrow d})y,y)\}\colon[1]\mathbb{N}}\,\text{Part A}}{x\colon[0]\mathbb{N},y\colon[1]\mathbb{N}\vdash\{0\mapsto add((*^{\to d}_{\leftarrow c})x,x),1\mapsto y\}\colon\{0\mapsto\mathbb{N},1\mapsto\mathbb{N}\}\ \rule[-0.15em]{0.25em}{0.9em}\ x\colon[0]\mathbb{N},y\colon[1]\mathbb{N}\vdash\{0\mapsto x,1\mapsto add((*^{\to c}_{\leftarrow d})y,y)\}\colon\{0\mapsto\mathbb{N},1\mapsto\mathbb{N}\}}\,\text{Part B}}{x\colon[0]\mathbb{N},y\colon[1]\mathbb{N}\vdash\{0\mapsto[add((*^{\to d}_{\leftarrow c})x,x),x],1\mapsto[add((*^{\to c}_{\leftarrow d})y,y),y]\}\colon\{0\mapsto\mathbb{N},1\mapsto\mathbb{N}\}}\,\mathrm{EC}$$

Figure 3.6: A typed global term that solves the exclusive-or problem (Example 3.3.1).

use the following auxiliary notion. An *investigator* is a partial map from processes to program variables.

For a typed hyperterm $\mathcal{O}$, we will give $[\![\mathcal{O}]\!]$, which is a tuple of programs indexed by $\mathbb{P}$. Also, we define $(\!|\mathcal{O}|\!)$ at the same time as $[\![\mathcal{O}]\!]$, where $(\!|\mathcal{O}|\!)$ is a sequence of investigators. The length of $(\!|\mathcal{O}|\!)$ is the same as that of $\mathcal{O}$. We refer to the last element of $(\!|\mathcal{O} \mid M|\!)$ as $(\!|\mathcal{O} \mid \overline{M}|\!)$, the second to last element of $(\!|\mathcal{O} \mid M \mid N|\!)$ as $(\!|\mathcal{O} \mid \overline{M} \mid N|\!)$ and so on.

Example 3.4.4 (Specifying elements of a sequence) *Suppose* $(\!|M^+ \mid N^+|\!) = \{0 \mapsto \boldsymbol{x}, 1 \mapsto \boldsymbol{y}\} \mid \{0 \mapsto \boldsymbol{z}\}$*. Then,* $(\!|M^+ \mid \overline{N^+}|\!) = \{0 \mapsto \boldsymbol{z}\}$ *and thus* $(\!|M^+ \mid \overline{N^+}|\!)(0) = \boldsymbol{z}$*.*

We let ϵ denote $(p_i = \epsilon)_{i\in\mathbb{P}}$. Also, $(p_i)_{i\in\mathbb{P}}; (q_i)_{i\in\mathbb{P}}$ denotes $(p_i; q_i)_{i\in\mathbb{P}}$. And $(p)_j$ denotes $(q_i)_{i\in\mathbb{P}}$ where $q_j = p$ and $q_i = \epsilon$ for all $i \neq j$. The definition is inductive over the type derivations. What we do below is essentially compilation from lambda terms to imperative programs[9]. After three pages, there is an example showing compilation of a concrete λ-GD global term.

ij**-com**

$$\begin{aligned}
&[\![\mathcal{O}_0 \mid \mathcal{O}_1 \mid \{i \mapsto (*\substack{\rightarrow d\\ \leftarrow c})M\} \mid \{j \mapsto (*\substack{\rightarrow c\\ \leftarrow d})N\}]\!]\\
=&[\![\mathcal{O}_0 \mid \{i \mapsto M\}]\!]; [\![\mathcal{O}_1 \mid \{j \mapsto N\}]\!];\\
&(d \leftarrow (\!|\mathcal{O}_0 \mid \overline{\{i \mapsto M\}}|\!)(i); \mathtt{x} \leftarrow c)_i;\\
&(c \leftarrow (\!|\mathcal{O}_1 \mid \overline{\{j \mapsto N\}}|\!)(j); \mathtt{y} \leftarrow d)_j \ ,
\end{aligned}$$

$$\begin{aligned}
&(\!|\mathcal{O}_0 \mid \mathcal{O}_1 \mid \{i \mapsto (*\substack{\rightarrow d\\ \leftarrow c})M\} \mid \{j \mapsto (*\substack{\rightarrow c\\ \leftarrow d})N\}|\!)\\
=&(\!|\overline{\mathcal{O}_0} \mid \{i \mapsto M\}|\!) \mid (\!|\overline{\mathcal{O}_1} \mid \{j \mapsto N\}|\!) \mid\\
&\{j \mapsto \mathtt{y}\} \mid \{i \mapsto \mathtt{x}\}
\end{aligned}$$

where `x` and `y` are new variables not used in induction hypotheses (we omit writing this restriction on each case below),

EW

$$[\![\mathcal{O} \mid \mathsf{abort}]\!] = [\![\mathcal{O}]\!] \ , \quad (\!|\mathcal{O} \mid \mathsf{abort}|\!) = (\!|\mathcal{O}|\!) \mid \emptyset \ ,$$

[9]That is one reason to try implementing it using Haskell in Chapter 5 because the Haskell compiler takes care of lambda-to-imperative translation.

EC

$$\begin{aligned}
&\llbracket \mathcal{O} \mid ([M_i, N_i])_{i\in I} \rrbracket \\
=&\llbracket \mathcal{O} \mid (M_i) \mid (N_i) \rrbracket; \\
&\Big(\mathtt{k}_i \leftarrow \llparenthesis \mathcal{O} \mid \overline{(M_i)} \mid (N_i) \rrparenthesis(i); \mathtt{k}_i \leftarrow \llparenthesis \mathcal{O} \mid (M_i) \mid \overline{(N_i)} \rrparenthesis(i)\Big)_{i\in I},
\end{aligned}$$

$$\llparenthesis \mathcal{O} \mid ([M_i, N_i])_{i\in I} \rrparenthesis = \llparenthesis \overline{\mathcal{O}} \mid ([M_i, N_i])_{i\in I} \rrparenthesis \mid \{i \mapsto \mathtt{k}_i\}_{i\in I}$$

EE

$$\begin{aligned}
\llbracket \mathcal{O} \mid N^+ \mid M^+ \mid \mathcal{O}' \rrbracket =& \llbracket \overline{\mathcal{O}} \mid N^+ \mid M^+ \mid \mathcal{O}' \rrbracket \mid \\
& \llbracket \mathcal{O} \mid N^+ \mid \overline{M^+} \mid \mathcal{O}' \rrbracket \mid \\
& \llbracket \mathcal{O} \mid \overline{N^+} \mid M^+ \mid \mathcal{O}' \rrbracket \mid \\
& \llbracket \mathcal{O} \mid N^+ \mid M^+ \mid \overline{\mathcal{O}'} \rrbracket \ , \\
\llparenthesis \mathcal{O} \mid N^+ \mid M^+ \mid \mathcal{O}' \rrparenthesis =& \llparenthesis \overline{\mathcal{O}} \mid N^+ \mid M^+ \mid \mathcal{O}' \rrparenthesis \mid \\
& \llparenthesis \mathcal{O} \mid N^+ \mid \overline{M^+} \mid \mathcal{O}' \rrparenthesis \mid \\
& \llparenthesis \mathcal{O} \mid \overline{N^+} \mid M^+ \mid \mathcal{O}' \rrparenthesis \mid \\
& \llparenthesis \mathcal{O} \mid N^+ \mid M^+ \mid \overline{\mathcal{O}'} \rrparenthesis \ ,
\end{aligned}$$

IC

$$\begin{aligned}
\llbracket \mathcal{O} \mid M^+[x/y] \rrbracket &= \llbracket \overline{\mathcal{O}} \mid M^+ \rrbracket \mid \llbracket \mathcal{O} \mid \overline{M^+} \rrbracket[\mathtt{x}_x/\mathtt{x}_y] \\
\llparenthesis \mathcal{O} \mid M^+[x/y] \rrparenthesis &= \llparenthesis \overline{\mathcal{O}} \mid M^+ \rrparenthesis \mid \llparenthesis \mathcal{O} \mid \overline{M^+} \rrparenthesis[\mathtt{x}_x/\mathtt{x}_y]
\end{aligned}$$

$[i]$**Ax**

$$\begin{aligned}
\llbracket x \rrbracket =& \epsilon \ , \\
\llparenthesis x \rrparenthesis =& \{i \mapsto \mathtt{i}_x\} \ ,
\end{aligned}$$

$[i]\bot\mathcal{E}$

$$\begin{aligned}
\llbracket \mathcal{O} \mid \{i \mapsto \mathsf{abort}\,\} \rrbracket =& \llbracket \mathcal{O} \mid \{i \mapsto M\} \rrbracket; (\mathtt{x} \leftarrow \epsilon)_i \ , \\
\llparenthesis \mathcal{O} \mid \{i \mapsto \mathsf{abort}\,\} \rrparenthesis =& \llparenthesis \overline{\mathcal{O}} \mid \{i \mapsto M\} \rrparenthesis \mid \{i \mapsto \mathtt{x}\} \ ,
\end{aligned}$$

$[i] \supset \mathcal{I}$

$$\begin{aligned}
\llbracket \mathcal{O} \mid \{i \mapsto \lambda x.M\} \rrbracket =& \llbracket \mathcal{O} \mid \{i \mapsto M\} \rrbracket; \Big(\mathtt{z} \leftarrow \lambda x. \llparenthesis \mathcal{O} \mid \overline{\{i \mapsto M\}} \rrparenthesis(i)\Big)_i \ , \\
\llparenthesis \mathcal{O} \mid \{i \mapsto \lambda x.M\} \rrparenthesis =& \llparenthesis \overline{\mathcal{O}} \mid \{i \mapsto M\} \rrparenthesis \mid \{i \mapsto \mathtt{z}\} \ ,
\end{aligned}$$

$[i] \supset \mathcal{E}$

$$\begin{aligned}
\llbracket \mathcal{O}_0 \mid \mathcal{O}_1 \mid \{i \mapsto MN\} \rrbracket =& \llbracket \mathcal{O}_0 \mid \{i \mapsto M\} \rrbracket ; \llbracket \mathcal{O}_1 \mid \{i \mapsto N\} \rrbracket ; \\
& \big(\mathtt{z} \leftarrow \llparenthesis \mathcal{O}_0 \mid \overline{M} \rrparenthesis (i)\ \llparenthesis \mathcal{O}_1 \mid \overline{N} \rrparenthesis (i)\big)_i \ , \\
\llparenthesis \mathcal{O}_0 \mid \mathcal{O}_1 \mid \{i \mapsto MN\} \rrparenthesis =& \llparenthesis \overline{\mathcal{O}_0} \mid \{i \mapsto M\} \rrparenthesis \mid \llparenthesis \overline{\mathcal{O}_1} \mid \{i \mapsto N\} \rrparenthesis \mid \{i \mapsto \mathtt{z}\} \ ,
\end{aligned}$$

$[i] \wedge \mathcal{E}_0$

$$\begin{aligned}
\llbracket \mathcal{O} \mid \{i \mapsto \pi_{\mathsf{l}}\,(M)\} \rrbracket =& \llbracket \mathcal{O} \mid \{i \mapsto M\} \rrbracket ; \Big(\mathtt{z} \leftarrow \pi_{\mathsf{l}} \Big(\llparenthesis \mathcal{O} \mid \overline{\{i \mapsto M\}} \rrparenthesis (i)\Big)\Big)_i \ , \\
\llparenthesis \mathcal{O} \mid \{i \mapsto \pi_{\mathsf{l}}\,(M)\} \rrparenthesis =& \llparenthesis \overline{\mathcal{O}} \mid \{i \mapsto M\} \rrparenthesis \mid \{i \mapsto \mathtt{z}\} \ ,
\end{aligned}$$

$[i] \wedge \mathcal{E}_1$

$$\begin{aligned}
\llbracket \mathcal{O} \mid \{i \mapsto \pi_{\mathsf{r}}\,(M)\} \rrbracket =& \llbracket \mathcal{O} \mid \{i \mapsto M\} \rrbracket ; \Big(\mathtt{z} \leftarrow \pi_{\mathsf{r}} \Big(\llparenthesis \mathcal{O} \mid \overline{\{i \mapsto M\}} \rrparenthesis (i)\Big)\Big)_i \ , \\
\llparenthesis \mathcal{O} \mid \{i \mapsto \pi_{\mathsf{r}}\,(M)\} \rrparenthesis =& \llparenthesis \overline{\mathcal{O}} \mid \{i \mapsto M\} \rrparenthesis \mid \{i \mapsto \mathtt{z}\} \ ,
\end{aligned}$$

$[i] \vee \mathcal{I}_0$

$$\begin{aligned}
\llbracket \mathcal{O} \mid \{i \mapsto \mathsf{inl}\,(M)\} \rrbracket =& \llbracket \mathcal{O} \mid \{i \mapsto M\} \rrbracket ; \Big(\mathtt{z} \leftarrow \mathsf{inl} \Big(\llparenthesis \mathcal{O} \mid \overline{\{i \mapsto M\}} \rrparenthesis (i)\Big)\Big)_i \ , \\
\llparenthesis \mathcal{O} \mid \{i \mapsto \mathsf{inl}\,(M)\} \rrparenthesis =& \llparenthesis \overline{\mathcal{O}} \mid \{i \mapsto M\} \rrparenthesis \mid \{i \mapsto \mathtt{z}\} \ ,
\end{aligned}$$

$[i] \vee \mathcal{I}_1$

$$\begin{aligned}
\llbracket \mathcal{O} \mid \{i \mapsto \mathsf{inr}\,(M)\} \rrbracket =& \llbracket \mathcal{O} \mid \{i \mapsto M\} \rrbracket ; \Big(\mathtt{z} \leftarrow \mathsf{inr} \Big(\llparenthesis \mathcal{O} \mid \overline{\{i \mapsto M\}} \rrparenthesis (i)\Big)\Big)_i \ , \\
\llparenthesis \mathcal{O} \mid \{i \mapsto \mathsf{inr}\,(M)\} \rrparenthesis =& \llparenthesis \overline{\mathcal{O}} \mid \{i \mapsto M\} \rrparenthesis \mid \{i \mapsto \mathtt{z}\} \ ,
\end{aligned}$$

$[i] \vee \mathcal{E}$

$$\begin{aligned}
& \llbracket \mathcal{O}_0 \mid \mathcal{O}_1 \mid \mathcal{O}_2 \mid \{i \mapsto \mathsf{match}\, M \,\mathsf{of}\, \mathsf{inl}(x).N_0/\mathsf{inr}(y).N_1\} \rrbracket \\
=& \llbracket \mathcal{O}_0 \mid \{i \mapsto M\} \rrbracket ; \\
& \Big(\mathtt{x}_x \leftarrow \mathsf{match}\, \llparenthesis \mathcal{O}_0 \mid \overline{\{i \mapsto M\}} \rrparenthesis (i) \,\mathsf{of}\, \mathsf{inl}(z).z/\mathsf{inr}(w).\epsilon\Big)_i ; \llbracket \mathcal{O}_1 \mid \{i \mapsto N_0\} \rrbracket ; \\
& \Big(\mathtt{x}_y \leftarrow \mathsf{match}\, \llparenthesis \mathcal{O}_0 \mid \overline{\{i \mapsto M\}} \rrparenthesis (i) \,\mathsf{of}\, \mathsf{inl}(z).\epsilon/\mathsf{inr}(w).w\Big)_i ; \llbracket \mathcal{O}_2 \mid \{i \mapsto N_1\} \rrbracket ; \\
& \Big(\mathtt{k} \leftarrow \mathsf{match} \llparenthesis \mathcal{O}_0 \mid \overline{\{i \mapsto M\}} \rrparenthesis (i) \mathsf{of} \\
& \qquad \mathsf{inl}(z).\llparenthesis \mathcal{O}_1 \mid \overline{\{i \mapsto N_0\}} \rrparenthesis (i)/\mathsf{inr}(w).\llparenthesis \mathcal{O}_2 \mid \overline{\{i \mapsto N_1\}} \rrparenthesis (i)\Big)_i \ ,
\end{aligned}$$

$$\begin{aligned}
& \llparenthesis \mathcal{O}_0 \mid \mathcal{O}_1 \mid \mathcal{O}_2 \mid \{i \mapsto \mathsf{match}\, M \,\mathsf{of}\, \mathsf{inl}(x).N_0/\mathsf{inr}(y).N_1\} \rrparenthesis \\
=& \llparenthesis \overline{\mathcal{O}_0} \mid \{i \mapsto M\} \rrparenthesis \mid \llparenthesis \overline{\mathcal{O}_1} \mid \{i \mapsto N_0\} \rrparenthesis \mid \llparenthesis \overline{\mathcal{O}_2} \mid \{i \mapsto N_1\} \rrparenthesis \mid \{i \mapsto \mathtt{k}\} \ .
\end{aligned}$$

The rules IE and IW do not appear above because these rules do not change the form of global terms. When $(x_i : [i]\iota_i)_{i\in\mathbb{P}} \vdash M : (\bigwedge\!\!\!\bigwedge_{i\in\mathbb{P}}[i]o_i)$ is derivable, we can define a typed protocol using the above translation. We set $\mathtt{i}_i$ to be $\mathtt{x}_{x_i}$, $\mathtt{o}_i$ to be arbitrarily chosen fresh program variables, L to be the set of locations occurring in the derivation, we set the family of programs to be $\llbracket M\rrbracket ; (\mathtt{o}_i \leftarrow \pi_i(\llparenthesis \overline{M} \rrparenthesis(i)))_{i\in\mathbb{P}}$, where π_i is obtained by composing i times π_{r} to π_{l}. We set g, d, t_i accordingly so that the program is typed. This concludes the translation.

Example 3.4.5 (Translation of a typed term into a typed protocol) *The typing derivation in Figure 3.6 can be translated as follows. We follow the derivation from top to bottom. First we look at the top sequents and translate the typed terms:*

$$\begin{aligned}
&\llbracket\{0 \mapsto x\}\rrbracket = \epsilon\\
&\llparenthesis\{0 \mapsto x\}\rrparenthesis = \{0 \mapsto \boldsymbol{i}_x\}\\
&\llbracket\{1 \mapsto y\}\rrbracket = \epsilon\\
&\llparenthesis\{1 \mapsto y\}\rrparenthesis = \{1 \mapsto \boldsymbol{i}_y\}\ .
\end{aligned}$$

Next we look one step[10] *below:*

$$\begin{aligned}
&\llbracket\{0 \mapsto (*^{\to d}_{\leftarrow c})x\} \ \boldsymbol{|}\ \{1 \mapsto (*^{\to c}_{\leftarrow d})y\}\rrbracket = (d \leftarrow \boldsymbol{i}_x ; \boldsymbol{y} \leftarrow c)_0 ; (c \leftarrow \boldsymbol{i}_y ; \boldsymbol{x} \leftarrow d)_1\\
&\llparenthesis\{0 \mapsto (*^{\to d}_{\leftarrow c})x\} \ \boldsymbol{|}\ \{1 \mapsto (*^{\to c}_{\leftarrow d})y\}\rrparenthesis = \{0 \mapsto \boldsymbol{y}\} \ \boldsymbol{|}\ \{1 \mapsto \boldsymbol{x}\}\ .
\end{aligned}$$

And so on:

$$\begin{aligned}
&\llbracket\{0 \mapsto add((*^{\to d}_{\leftarrow c})x, x)\} \ \boldsymbol{|}\ \{1 \mapsto add((*^{\to c}_{\leftarrow d})y, y)\}\rrbracket\\
&\ = (d \leftarrow \boldsymbol{i}_x ; \boldsymbol{y} \leftarrow c)_0 ; (c \leftarrow \boldsymbol{i}_y ; \boldsymbol{x} \leftarrow d)_1 ; (\boldsymbol{z} \leftarrow add(\boldsymbol{y}, \boldsymbol{i}_x))_0 ; (\boldsymbol{w} \leftarrow add(\boldsymbol{x}, \boldsymbol{i}_y))_1\\
&\llparenthesis\{0 \mapsto add((*^{\to d}_{\leftarrow c})x, x)\} \ \boldsymbol{|}\ \{1 \mapsto add((*^{\to c}_{\leftarrow d})y, y)\}\rrparenthesis\\
&\ = \{0 \mapsto \boldsymbol{z}\} \ \boldsymbol{|}\ \{1 \mapsto \boldsymbol{x}\}\ .
\end{aligned}$$

$$\begin{aligned}
&\llbracket\{0 \mapsto add((*^{\to d}_{\leftarrow c})x, x), 1 \mapsto y\} \ \boldsymbol{|}\ \{0 \mapsto x, 1 \mapsto add((*^{\to c}_{\leftarrow d})y, y)\}\rrbracket\\
&= (d \leftarrow \boldsymbol{i}_x ; \boldsymbol{y} \leftarrow c)_0 ; (c \leftarrow \boldsymbol{i}_y ; \boldsymbol{x} \leftarrow d)_1 ; (\boldsymbol{z} \leftarrow add(\boldsymbol{y}, \boldsymbol{i}_x))_0 ; (\boldsymbol{w} \leftarrow add(\boldsymbol{x}, \boldsymbol{i}_y))_1 ;\\
&\quad (\boldsymbol{x'} \leftarrow \boldsymbol{i}_x)_0 ; (\boldsymbol{y'} \leftarrow \boldsymbol{i}_y)_1\\
&\llparenthesis\{0 \mapsto add((*^{\to d}_{\leftarrow c})x, x), 1 \mapsto y\} \ \boldsymbol{|}\ \{0 \mapsto x, 1 \mapsto add((*^{\to c}_{\leftarrow d})y, y)\}\rrparenthesis\\
&= \{0 \mapsto \boldsymbol{z}, 1 \mapsto \boldsymbol{y'}\} \ \boldsymbol{|}\ \{0 \mapsto \boldsymbol{x'}, 1 \mapsto \boldsymbol{w}\}\ .
\end{aligned}$$

[10] A step in the type derivation is translated into a pair of equalities with different kinds of parentheses.

$$
\begin{aligned}
&[\![\{0 \mapsto [add((*^{\to d}_{\leftarrow c})x, x), x], 1 \mapsto [add((*^{\to c}_{\leftarrow d})y, y), y]\}]\!]\\
&= (d \leftarrow \boldsymbol{i}_x; \boldsymbol{y} \leftarrow c)_0\,; (c \leftarrow \boldsymbol{i}_y; \boldsymbol{x} \leftarrow d)_1\,; (\boldsymbol{z} \leftarrow add(\boldsymbol{y}, \boldsymbol{i}_x))_0\,; (\boldsymbol{w} \leftarrow add(\boldsymbol{x}, \boldsymbol{i}_y))_1\,;\\
&\quad (\boldsymbol{x'} \leftarrow \boldsymbol{i}_x)_0\,; (\boldsymbol{y'} \leftarrow \boldsymbol{i}_y)_1\,; (\boldsymbol{u} \leftarrow \boldsymbol{z}; \boldsymbol{u} \leftarrow \boldsymbol{x'})_0\,; (\boldsymbol{v} \leftarrow \boldsymbol{w}; \boldsymbol{v} \leftarrow \boldsymbol{y'})_1\\
&(\!|\{0 \mapsto [add((*^{\to d}_{\leftarrow c})x, x), x], 1 \mapsto [add((*^{\to c}_{\leftarrow d})y, y), y]\}|\!)\\
&= \{0 \mapsto \boldsymbol{u}, 1 \mapsto \boldsymbol{v}\} \enspace .
\end{aligned}
$$

The last two clauses result in a typed protocol that solves the exclusive-or problem (Example 3.3.1) for the same reason as the typed protocol in Example 3.3.13.

Given that the translation solves a typed input-output problem, we have to make sure that the original global term solves the same problem. Otherwise, the translation incorrectly produced a working typed protocol from not-working a global term. We have to show that the translation is correct in the following sense.

Proposition 3.4.6 *For a derivable typed hypersequent $\mathcal{O}$, let $(\!|\mathcal{O}|\!) = f_0 \mid f_1 \mid \cdots \mid f_n$ so that each f_j is a partial map from processes in $\mathbb{P}$ to program variables. Assume that there is an execution of $[\![\mathcal{O}]\!]$ whose final system snapshot $\langle \vec{s}, \vec{v} \rangle$ satisfies $s_{f_j(i)} = V_{ji}$ for each $i \in \mathrm{dom}(f_j)$. Then, there exists a reduction relation*

$$
(\{\}, \mathcal{O}) \rightsquigarrow^* (S, \{i \mapsto V_{0i}\}_{i \in \mathrm{dom}(f_0)} \mid \cdots \mid \{i \mapsto V_{ni}\}_{i \in \mathrm{dom}(f_n)})
$$

for some store S.

Proof By induction on the typing derivation for $\mathcal{O}$. All cases are more or less straightforward, of which the most complicated cases are ij-com rule and the EC rule.

ij**-com** Assume that there is an execution of

$$
\begin{aligned}
&[\![\mathcal{O}_0 \mid \{i \mapsto M\}]\!]; [\![\mathcal{O}_1 \mid \{j \mapsto N\}]\!];\\
&(d \leftarrow (\!|\mathcal{O}_0 \mid \overline{\{i \mapsto M\}}|\!)(i); \mathtt{x} \leftarrow c)_i;\\
&(c \leftarrow (\!|\mathcal{O}_1 \mid \overline{\{j \mapsto N\}}|\!)(j); \mathtt{y} \leftarrow d)_j
\end{aligned}
$$

whose final system snapshot contains process snapshots s_l for each process $l \in \mathbb{P}$ that map program variables as:

$$
\begin{aligned}
(\!|\overline{\mathcal{O}_0} \mid \{i \mapsto M\}|\!) &\mapsto (V_{kl})_{0 \le k < |\mathcal{O}_0|}\\
(\!|\overline{\mathcal{O}_1} \mid \{j \mapsto N\}|\!) &\mapsto (V_{kl})_{|\mathcal{O}_0| \le k < |\mathcal{O}_0| + |\mathcal{O}_1|} \enspace .
\end{aligned}
$$

Moreover, we assume $s_i(\mathtt{x}) = V_{|\mathcal{O}_0|+|\mathcal{O}_1|+1,i}$ and $s_j(\mathtt{y}) = V_{|\mathcal{O}_0|+|\mathcal{O}_1|,j}$. In order to use the induction hypotheses, we have to transform our assumptions to the assumptions needed by the induction hypotheses on the branches of ij-com rule.

By the form of the programs, there exists an execution of $[\![\mathcal{O}_0 \mid \{i \mapsto M\}]\!]$ whose final system snapshot contains a process snapshot s'_p for each process $p \in \mathbb{P}$ that maps program variables as:

$$\llparenthesis \overline{\mathcal{O}_0} \mid \{i \mapsto M\} \rrparenthesis \mapsto (V_{kp})_{0 \le k < |\mathcal{O}_0|}$$

and moreover s'_i maps $\llparenthesis \mathcal{O}_0 \mid \overline{\{i \mapsto M\}} \rrparenthesis$ to V where V is equal to $V_{|\mathcal{O}_0|+|\mathcal{O}_1|+1,i}$ unless $V_{|\mathcal{O}_0|+|\mathcal{O}_1|+1,i}$ is ϵ. Similarly, there exists an execution of $[\![\mathcal{O}_1 \mid \{j \mapsto N\}]\!]$ whose final system snapshot contains a process snapshot s''_p for each $p \in \mathbb{P}$ that maps program variables as:

$$\llparenthesis \overline{\mathcal{O}_1} \mid \{j \mapsto N\} \rrparenthesis \mapsto (V_{kp})_{|\mathcal{O}_0| \le k < |\mathcal{O}_0|+|\mathcal{O}_1|}$$

and moreover s''_j maps $\llparenthesis \mathcal{O}_1 \mid \overline{\{j \mapsto N\}} \rrparenthesis$ to W where W is equal to $V_{|\mathcal{O}_0|+|\mathcal{O}_1|,j}$ unless $V_{|\mathcal{O}_0|+|\mathcal{O}_1|,j}$ is ϵ.

By the induction hypothesis on $\mathcal{O}_0 \mid \{i \mapsto M\}$, there exists a reduction relation

$$\begin{aligned}&(\{\}, \mathcal{O}_0 \mid \{i \mapsto M\})\\ \rightsquigarrow&(S', \mathop{\Big|}_{0 \le k < |\mathcal{O}_0|} \left(M_k^+\right) \mid \{i \mapsto V\})\end{aligned}$$

for some store S' and $M_k^+(p) = V_{kp}$ for each $p \in \mathbb{P}$. Also, by the induction hypothesis on $\mathcal{O}_1 \mid \{j \mapsto N\}$, there exists a reduction relation

$$\begin{aligned}&(\{\}, \mathcal{O}_1 \mid \{j \mapsto N\})\\ \rightsquigarrow&(S'', \mathop{\Big|}_{|\mathcal{O}_0| \le k < |\mathcal{O}_0|+|\mathcal{O}_1|} \left(M_k^+\right) \mid \{j \mapsto W\})\end{aligned}$$

for some store S'' and $M_k^+(p) = V_{kp}$ for each $p \in \mathbb{P}$.

When we combine these induction hypotheses, we obtain

$$\begin{aligned}&(\{\}, \mathcal{O}_0 \mid \mathcal{O}_1 \mid \{i \mapsto (*^{\rightarrow d}_{\leftarrow c})M\} \mid \{j \mapsto (*^{\rightarrow c}_{\leftarrow d})N\})\\ \rightsquigarrow^*&(S' \sqcup S'', (M_k^+)_{0 \le k < |\mathcal{O}_0|+|\mathcal{O}_1|} \mid \{i \mapsto (*^{\rightarrow d}_{\leftarrow c})V\} \mid \{j \mapsto (*^{\rightarrow c}_{\leftarrow d})W\})\\ \rightsquigarrow^*&(S, (M_k^+)_{0 \le k < |\mathcal{O}_0|+|\mathcal{O}_1|} \mid \{i \mapsto V_{|\mathcal{O}_0|+|\mathcal{O}_1|+1,i}\} \mid \{j \mapsto V_{|\mathcal{O}_0|+|\mathcal{O}_1|,j}\})\ .\end{aligned}$$

for some S.

EC Assume that there exists an execution of

$$\begin{aligned}&[\![\mathcal{O} \mid (M_i)_{i \in I} \mid (N_i)_{i \in I}]\!];\\ &\Big(\mathtt{k}_i \leftarrow \llparenthesis \mathcal{O} \mid \overline{(M_i)} \mid (N_i) \rrparenthesis(i); \mathtt{k}_i \leftarrow \llparenthesis \mathcal{O} \mid (M_i) \mid \overline{(N_i)} \rrparenthesis(i)\Big)_{i \in I}\end{aligned}$$

whose final system snapshot is $\langle \overrightarrow{s}, \overrightarrow{v} \rangle$ where for each process $p \in \mathbb{P}$, s_p maps

$$(\!| \overline{\mathcal{O}} \mathbin{\|} ([M'_i, N'_i])_{i \in I} |\!) \mapsto (V_{kp})_{0 \le k < |\mathcal{O}|}$$

and moreover $s_i(\mathtt{k}_i)$ is $V_{|\mathcal{O}|,i}$ for each $i \in I$.

By the form of the programs, there exists an execution of $[\![\mathcal{O} \mathbin{\|} (M_i)_{i\in I} \mathbin{\|} (N_i)_{i \in I}]\!]$ whose final system snapshot $\langle \overrightarrow{s'}, \overrightarrow{v'} \rangle$ satisfies

- for any $i \in I$ with $\langle M_i, N_i \rangle = \langle M'_i, N'_i \rangle$, the process snapshot s'_i maps $(\!| \mathcal{O} \mathbin{\|} \overline{(M_i)_{i\in I}} \mathbin{\|} (N_i)_{i\in I} |\!)$ to V_i and $(\!| \mathcal{O} \mathbin{\|} (M_i)_{i\in I} \mathbin{\|} \overline{(N_i)_{i\in I}} |\!)$ to W_i where V_i is identical to $V_{|\mathcal{O}|,i}$ or ϵ and W_i is identical to $V_{|\mathcal{O}|,i}$;
- for any $i \in I$ with $\langle M_i, N_i \rangle = \langle N'_i, M'_i \rangle$, the process snapshot s'_i maps $(\!| \mathcal{O} \mathbin{\|} \overline{(M_i)_{i\in I}} \mathbin{\|} (N_i)_{i\in I} |\!)$ to V_i and $(\!| \mathcal{O} \mathbin{\|} (M_i)_{i\in I} \mathbin{\|} \overline{(N_i)_{i\in I}} |\!)$ to W_i where W_i is identical to $V_{|\mathcal{O}|,i}$ or ϵ and V_i is identical to $V_{|\mathcal{O}|,i}$; and
- for any $p \in \mathbb{P}$, s'_p maps $(\!| \overline{\mathcal{O}} \mathbin{\|} (M_i)_{i\in I} \mathbin{\|} (N_i)_{i\in I} |\!)$ to $(V_{kp})_{0 \le p < |\mathcal{O}|}$.

By the induction hypothesis, there exists a reduction relation

$$\begin{aligned} &(\{\}, \mathcal{O} \mathbin{\|} (M_i)_{i\in I} \mathbin{\|} (N_i)_{i\in I}) \\ \rightsquigarrow^* &(S, \mathop{\|}_{0 \le k < |\mathcal{O}|} ({M_k}^+) \mathbin{\|} (V_i)_{i\in I} \mathbin{\|} (W_i)_{i \in I}) \end{aligned}$$

for some store S, global terms ${M_k}^+$ satisfying ${M_k}^+(p) = V_{kp}$ for each $p \in \mathbb{P}$. Thus, there is also a reduction relation

$$\begin{aligned} &(\{\}, \mathcal{O} \mathbin{\|} ([M'_i, N'_i])_{i\in I}) \\ \rightsquigarrow^* &(S, \mathop{\|}_{0 \le k < |\mathcal{O}|} ({M_k}^+) \mathbin{\|} (V_{|\mathcal{O}|,i})_{i\in I}) \ . \end{aligned}$$ ■

Proof (of Theorem 3.4.3) Assume that a global term M^+ solves a typed input-output problem R. We claim that the typed protocol $[\![M^+]\!]; (\mathtt{o}_i \leftarrow (\!|M^+|\!)(i))_{i \in \mathbb{P}}$ solves R. By Proposition 3.4.6, if there is an execution of $[\![M^+]\!][N_i/\mathtt{i}_i]$ that yields the content of $(\!|M^+|\!)(i)$ to be V_i for each $i \in \mathbb{P}$, the reduction relation $(\{\}, M^+[N_i/x_i]) \rightsquigarrow (S, \{i \mapsto V_i\}_{i\in\mathbb{P}})$ holds for some store S. Since the global term M^+ solves R, the pair $\langle (N_i)_{i\in\mathbb{P}}, (V_i)_{i\in\mathbb{P}} \rangle$ is compatible with R. ■

For the other direction, we can use a universal waitfree problem called the participating set problem. Since the class of waitfreely solvable problems is generated by the participating set problem and problem compositions. We just have to solve the universal problem using λ-GD.

Theorem 3.4.7 (Completeness of λ-GD with regard to waitfreedom) *If there exists a typed protocol that solves a typed input-output problem, the problem is solvable by a typed global term of λ-GD.*

Proof Herlihy and Shavit [72] showed that a solution to a finite repetition of the participating set problem solves any waitfreely solvable problem. Also, n-party participating problem can be solved by a tournament of the two-party participating set problem[11]. Since λ-GD can express problem composition, it suffices to show a λ-GD term solving the two-party participating set problem.

In the *participating set problem* [22], each process i receives an id c_i and returns a set of ids S_i. The outputs must satisfy (i) $c_i \in S_i$; (ii) either $S_i \subseteq S_j$ or $S_j \subseteq S_i$; and (iii) $S_i \subseteq S_j$ if $c_i \in S_j$ for any $i, j \in \mathbb{P}$. For two processes, $\langle S_0, S_1\rangle$ can be $\langle\{c_0\}, \{c_0, c_1\}\rangle$, $\langle\{c_0, c_1\}, \{c_1\}\rangle$ or $\langle\{c_0, c_1\}, \{c_0, c_1\}\rangle$. The point is that the processes do not know their own ids in advance but receive their own ids as inputs. This is why just hardcoding processes' outputs does not work.

We are going to encode the participating set problem in λ-GD. For this, we introduce a base type called Id for process id's. Let there be an injection that maps a natural number i to a constant $c_i \colon \mathsf{Id}$. The additional typing rules involving Id are as follows, where $\mathbb{N} = (\bot \supset \bot) \vee (\bot \supset \bot)$:

$$\frac{}{\vdash c_n \colon [i]\mathsf{Id}} \qquad\qquad \frac{\Gamma \vdash M_0 \colon [i]\mathsf{Id} \qquad \Gamma \vdash M_1 \colon [i]\mathsf{Id}}{\Gamma \vdash M_0 == M_1 \colon [i]\mathbb{N}} \ .$$

The additional reduction is

$$c_m == c_n \rightsquigarrow \begin{cases} \mathsf{inl}\,(\lambda x.x) & (\text{if } m = n) \\ \mathsf{inr}\,(\lambda x.x) & (\text{otherwise}) \ . \end{cases}$$

Also, $\mathsf{If}\, M \,\mathsf{then}\, N_0 \,\mathsf{else}\, N_1$ is an abbreviation for $\mathsf{match}\, M \,\mathsf{of}\, \mathsf{inl}(x).N_0/\mathsf{inr}(y).N_1$.

We represent a finite set of id's as a typed lambda term, whose type is $[i](\mathsf{Id} \supset \mathbb{B})$. Intuitively, a set takes an id and decides whether it is *in* or *out*. The emptyset is represented by a term $\lambda x.\mathsf{inr}\,(\bullet)$. When a finite set S is represented by a term M, the set $S \cup \{c\}$ is represented by a term $\lambda x.\,(\mathsf{If}\, x == c \,\mathsf{then}\, \mathsf{inl}\,(\bullet) \,\mathsf{else}\, Mx)$. With the above construction, we define abbreviations like $\{c_0, c_1, c_2\}$ although we do not equate $\{c_0, c_1, c_2\}$ and $\{c_0, c_2, c_1\}$.

Now, we are ready to construct a hyperterm solving the two-party participating

[11] To be exact, the construction for solving the n-party participating set problem using 2-party participating set problem is the same as a sorting network [16] for sorting n elements.

set problem. We can obtain a derivation of

$$x : [0]\mathsf{Id}, y : [1]\mathsf{Id} \vdash \{0 \mapsto [\{(*^{\rightarrow d}_{\leftarrow c})x, x\}, \{x\}], 1 \mapsto [\{(*^{\rightarrow c}_{\leftarrow d})y, y\}, \{y\}]\} :$$
$$0 \mapsto \mathsf{Id} \supset \mathbb{B}, 1 \mapsto \mathsf{Id} \supset \mathbb{B} \ .$$

One possible reduction sequence is as follows:

$$\begin{aligned}
&(\{\}, \{0 \mapsto [\{\underline{(*^{\rightarrow d}_{\leftarrow c})x}, x\}, \{x\}], 1 \mapsto [\{(*^{\rightarrow c}_{\leftarrow d})y, y\}, \{y\}]\}) \\
\rightsquigarrow_{\mathrm{W}} &(\{d \mapsto x\}, \{0 \mapsto [\{\underline{*_{\leftarrow c}}, x\}, \{x\}], 1 \mapsto [\{(*^{\rightarrow c}_{\leftarrow d})y, y\}, \{y\}]\}) \\
\rightsquigarrow_{\mathrm{R}} &(\{d \mapsto x\}, \{0 \mapsto [\{\mathsf{abort}\,, x\}, \{x\}], 1 \mapsto [\{\underline{(*^{\rightarrow c}_{\leftarrow d})y}, y\}, \{y\}]\}) \\
\rightsquigarrow_{\mathrm{W}} &(\{c \mapsto y, d \mapsto x\}, \{0 \mapsto [\{\mathsf{abort}\,, x\}, \{x\}], 1 \mapsto [\{\underline{*_{\leftarrow d}}, y\}, y], \{y\}\}) \\
\rightsquigarrow_{\mathrm{R}} &(\{c \mapsto y, d \mapsto x\}, \{0 \mapsto [\underline{\{\mathsf{abort}\,, x\}}, \{x\}], 1 \mapsto [\{x, y\}, \{y\}]\}) \\
\rightsquigarrow_{\mathrm{A}} &(\{c \mapsto y, d \mapsto x\}, \{0 \mapsto \underline{[\mathsf{abort}\,, \{x\}]}, 1 \mapsto [\{x, y\}, \{y\}]\}) \\
\rightsquigarrow_{\mathrm{A}} &(\{c \mapsto y, d \mapsto x\}, \{0 \mapsto \{x\}, 1 \mapsto \underline{[\{x, y\}, \{y\}]}\}) \\
\rightsquigarrow_{\mathrm{A}} &(\{c \mapsto y, d \mapsto x\}, \{0 \mapsto \{x\}, 1 \mapsto \{x, y\}\})
\end{aligned}$$

Moreover, the same initial configuration can reduce to

$$(\{c \mapsto y, d \mapsto x\}, \{0 \mapsto \{y, x\}, 1 \mapsto \{y\}\})$$

and

$$(\{c \mapsto y, d \mapsto x\}, \{0 \mapsto \{y, x\}, 1 \mapsto \{x, y\}\}) \ .$$

There are no other normal forms. These three normal forms correspond to the three allowed answers of the two-party participating set problem. ■

3.5 Related Work

Sonobe [129] gives sequent calculi for intermediate logics S_i and proved cut-elimination theorem for them. As a special case he gives S_ω, which coincides with Gödel-Dummett logic. The proof of cut-elimination is similar to that of Gentzen [58], involving the mix rule. No lambda calculi has been developed based on Sonobe's deduction system, probably because the deduction system involves a rule having unlimited number of assumptions.

Avron [8] formulates a hypersequent calculus for Gödel-Dummett logic and proves cut-elimination theorem using a method similar to Gentzen [58]. Also, he explains the intuition behind the communication rule as "the inputs through the ports in Γ_2' are transmitted to the component with output of type A_1. The inputs through Γ_1' are treated similarly." He did not mention the possibility of any transmission failures, which we exploited in order to characterize waitfreedom. Ciabattoni, Galatos and Terui [30] give a class of logics that have hypersequent calculi with cut-elimination. Their cut-elimination proof is general but it does not obviously reveal the computational content.

Baaz, Ciabattoni and Fermüller [11] propose a hypersequent-style natural deduction for Gödel-Dummett logic, but did not define reductions. Fermüller [48] gives a game semantics for Gödel-Dummett logic, which is based on Lorenzen game [130] and essentially proof searching bottom-to-up. Of course, there is a possibility of employing waitfree communication for proof searching. Indeed, Fermüller's dialogue game is sometimes forked and proceeds concurrently. However, he gives no explicit mention on waitfree computation.

Among numerous typed programming languages with parallelism, to our knowledge, none models waitfreedom. Abramsky [3]'s calculus PE_2 for classical linear logic is deterministic [3, Theorem 7.9] so that it is impossible to model waitfreedom using PE_2. The π-calculus [107], Join calculus [51], and even asynchronous π-calculus [80] have too strong synchronization abilities to model waitfreedom because a process can wait for an input.

Hirai [75, 76] compares the temporal order of waitfree computation and the Kripke models of a modal logic similar to Gödel-Dummett logic. The current work witnesses the constructive content of his model theoretic comparison.

3.6 Discussion

As a programming language, λ-GD allows efficient execution because it requires no synchronization among processes. Possibly, this calculus can be extended by synchronization primitives. It would be interesting to compare different synchronization primitives and different intermediate logics, generalizing waitfreedom and Gödel-Dummett logic. For example, it would be tedious but straightforward to adapt the hyper-lambda calculi here to the logics characterized by the Kripke frames of bounded width [29] because λ-GD is a special case of width 1. However, the author has no immediate idea on developing a general hyper-lambda calculi encompassing all logics with cut-eliminatable hypersequent calculi [30]. Some high-performance computation people suggested to study a weaker shared memory consistency than sequential consistency; that is, to remove the assumption that all read and write operations on shared memory are lined up in a temporal total order. In their realm, sequential consistency is considered too strong to assume everywhere. If we are to study a weaker shared memory consistency in the same approach as here, we have to study a weaker logic.

It will be worthwhile to develop a waitfree protocol verification mechanism in Coq because it is valuable to remove unnecessary synchronization while keeping the program correct in high performance computing.

An anonymous referee of FLOPS 2012 pointed out that the introduction of modalities $[i]$ is interesting on its own. We have not investigated the logical meaning of these modalities. We suspect these modalities are similar to nominals in the hybrid logic [21].

In λ-GD, the source of nondeterminism can be explicitly expressed as the store prophecy. If we can find a semantic counterpart Sch of the store prophecy, possibly, we can obtain a denotation $\mathcal{D}^{\mathsf{Sch}}$ of terms using a denotation $\mathcal{D}$ for normal forms. If that succeeds for classical logic, it will be interesting[12].

Nonetheless there is some more immediate future work. It would be better if we explicitly implement n-party participating set problem using λ-GD. Another remaining task is to formalize the strong normalization theorem formally in a proof assistant like Coq or Isabelle. Lindley and Stark [95] provided a concise proof of strong normalization of the natural deduction for intuitionistic propositional logic, containing disjunction. Lindley's proof was formalized in Nominal Isabelle by Doczkal and Schwinghammer [42].

Developing a second order formulation and performing the parametricity argument is another piece of future work. In Chapter 2, we studied an axiom $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$. Apart from this, we also tried developing a lambda calculus for $(\varphi \multimap \psi) \oplus (\psi \multimap \varphi)$ on top of a second-order formulation of intuitionistic linear logic in Chapter 4. While doing that, we noticed that adequacy of parametricity argument can be carried out using induction on just two type derivations; if we did the same for the second-order variant of λ-GD, it would require induction over many derivations.

Another immediate task is to verify cut-elimination theorem for the new communication rule presented here. Our com rule does not duplicate any contexts while the com' rule by Avron [8] duplicates the contexts Γ and Δ as

$$\frac{\mathcal{O}_0 \mathbin{\|} \Gamma, \Delta \vdash M : [i]\varphi \qquad \mathcal{O}_1 \mathbin{\|} \Gamma, \Delta \vdash N : [j]\psi}{\mathcal{O}_0 \mathbin{\|} \mathcal{O}_1 \mathbin{\|} \Gamma \vdash (*\overset{\rightarrow d}{\underset{\leftarrow c}{}})M : [i]\psi \mathbin{\|} \Delta \vdash (*\overset{\rightarrow c}{\underset{\leftarrow d}{}})N : [j]\varphi}\ .$$

Although Avron [8] showed cut-elimination theorem for com' rule, it is not clear whether the same theorem holds in our case.

[12]Kazushige Terui suggested the potential impact for classical logic.

3.7 Conclusions

We proposed λ-GD, a lambda calculus based on Avron's hypersequent calculus for Gödel-Dummett logic [8]. We proved normalization and non-abortfullness. The calculus characterizes the typed version of waitfree computation. Our result hints broader correspondence between proof theory and distributed computation.

Chapter 4

Prelinearity Axiom as an Asynchronous Communication Scheme

4.1 Summary

Danos and Krivine [38] state that disjunctive tautologies[1] such as the excluded middle $\varphi \vee (\varphi \supset \bot)$ and the symmetric excluded middle $(\varphi \supset \psi) \vee (\psi \supset \varphi)$ represent synchronization schemes. Indeed in their formalism, reduction of one term can wait for another term, thus making synchronization. However, one can find no communication among different concurrent elements of an executable. In this chapter, we show that logical axioms can specify how concurrent processes exchange information asynchronously.

We investigate the computational behavior of the prelinearity axiom $(\varphi \multimap \psi) \oplus (\psi \multimap \varphi)$ and left weakening on top of IMALL2, which is the fragment of second order intuitionistic linear logic with connectives $\{\forall, \multimap, \oplus\}$. In other words, our type system is based on the second-order monoidal t-norm logic (MTL) [46]. We interpret the disjunction $\oplus$ in the prelinearity axiom nondeterministically: whether one process can give information to the other or vice versa. In Section 4.2, where processes do not make synchronization, nondeterminism appears automatically as a slower process cannot pass information to a faster one. In Section 4.3, where processes make synchronization, we externally specify the nondeterministic choice. We adapt Danos and Krivine [38]'s realizability argument to our synchronous case and give simulation from the synchronous case to the asynchronous case.

The asynchronous semantics here is closely related to the hyper-lambda calculus for Gödel-Dummett logic presented in Hirai [75]. On top of intuitionistic logic, Gödel-

[1]A *tautology* is a theorem of classical propositional logic.

Dummett logic is axiomatized by Dummett axiom $(\varphi \supset \psi) \vee (\psi \supset \varphi)$, which is similar to the prelinearity axiom.

Our aim in this chapter is to reason about asynchronous communication on shared memory using parametricity argument. However, since processes can write into and read from shared memory, a straightforward approach would lead us to solving mixed-variant domain equations or using the step-index technique [4]. Although dealing with general references and parametricity is manageable [20], we choose a simpler, indirect approach. In Section 4.2 we deal with the operational semantics involving asynchronous communication using stores. In Section 4.3 we apply the classical realizability argument to an operational semantics where communication is made synchronously. In Sections 4.5 and 4.6, we discuss and conclude.

4.2 Asynchronous Semantics

We apply Danos and Krivine's classical realizability argument [38] to a lambda calculus involving communicating processes. Although the formulation is similar to [38], our formulation allows different processes to pass around lambda terms.

In this section, we give an abstract machine involving asynchronous shared memory communication, which is similar to the hyper-lambda calculus for Gödel-Dummett logic (Hirai [75]). We consider a programming language, which is a modification of Danos and Krivine [38]'s.

4.2.1 Dynamics

We assume a set PVar of propositional variables whose cardinality is countably infinite. We also assume countably infinitely many locations with involution $c \mapsto \bar{c}$ satisfying $\bar{c} \neq c$ and $\bar{\bar{c}} = c$. Metavariable c runs over locations. An *a-term* (asynchronous term) t is defined by BNF:

$$\begin{aligned} t ::= {} & x \mid (t)t \mid \lambda x.t \mid \mathsf{match}\, t\, \mathsf{of}\, \mathsf{inl}(x).t/\mathsf{inr}(x).t \mid \mathsf{inl}(t) \mid \mathsf{inr}(t) \mid \\ & (t \parallel t) \mid *^{\to c}_{\leftarrow c} \mid *_{\leftarrow c} \mid \mathsf{abort} \end{aligned}$$

where x is a variable and c is a location. The variable occurrences except in the first clause are binding. An *a-stack* (asynchronous stack) π is defined by a BNF:

$$\pi ::= \epsilon \mid t \cdot \pi \mid \langle \mathsf{inl}(x).(t, \pi)/\mathsf{inr}(x).(t, \pi) \rangle \ .$$

We write the set of a-terms as Λ_{a} and a-stacks Π_{a}. A *store* maps a location to an a-term, $\perp$ or $\top$. The *empty store* S_ϵ maps any location to $\perp$. For a store S, we define the updated store $S[c \mapsto x]$ to be the same as S except that $S[c \mapsto x](c)$ is x whatever $S(c)$ is. An *a-executable* (asynchronous executable) is a finite multiset on $\Lambda_{\mathsf{a}} \times \Pi_{\mathsf{a}}$, paired with a store.

We define a binary relation $\succ_{\mathsf{a}}$ on a-executables to be the smallest binary preorders that satisfy:

(cong) if $[t, \pi], S \succ_{\mathsf{a}} [t', \pi'], S'$ then $[t, \pi \parallel e], S \succ_{\mathsf{a}} [t', \pi' \parallel e], S'$;

(abort) $[\mathsf{abort}\,, \pi] \succ_{\mathsf{a}} \emptyset$;

(push) $[(t)u, \pi], S \succ_{\mathsf{a}} [t, u \cdot \pi], S$;

(store) $[\lambda x.t, u \cdot \pi], S \succ_{\mathsf{a}} [t[u/x], \pi], S$;

(ask) $[\mathsf{match}\, t\, \mathsf{of}\, \mathsf{inl}(x).u/\mathsf{inr}(y).v, \pi], S \succ_{\mathsf{a}} [t, \langle \mathsf{inl}(x).(u, \pi)/\mathsf{inr}(y).(v, \pi)\rangle], S$;

(ansL) $[\mathsf{inl}(v), \langle \mathsf{inl}(x).(t, \pi)/\mathsf{inr}(y).(u, \sigma)\rangle], S \succ_{\mathsf{a}} [t[v/x], \pi], S$;

(ansR) $[\mathsf{inr}(w), \langle \mathsf{inl}(x).(t, \pi)/\mathsf{inr}(y).(u, \sigma)\rangle], S \succ_{\mathsf{a}} [u[w/y], \sigma], S$;

(write) $[*_{\leftarrow c}^{\rightarrow \bar{c}}, t \cdot \pi], S[\bar{c} \mapsto \bot] \succ_{\mathsf{a}} [*_{\leftarrow c}, \pi], S[\bar{c} \mapsto t]$;

(write') $[*_{\leftarrow c}^{\rightarrow \bar{c}}, t \cdot \pi], S[\bar{c} \mapsto \top] \succ_{\mathsf{a}} [*_{\leftarrow c}, \pi], S[\bar{c} \mapsto \bot]$;

(read) $[*_{\leftarrow c}, \pi], S[c \mapsto u] \succ_{\mathsf{a}} [u, \pi], S[c \mapsto u]$;

(fail) $[*_{\leftarrow c}, \pi], S[c \mapsto \bot] \succ_{\mathsf{a}} \emptyset, S[c \mapsto \top]$;

(local-global) if $e_0, S_0 \succ_{\mathsf{a}} e_1, S_1$ then $e_0, S_0 \succ_{\mathsf{a}} e_1, S_1$; and

(dist) $[(t \parallel u), \pi \parallel e], S \succ_{\mathsf{a}} [t, \pi \parallel u, \pi \parallel e \parallel e], S$.

Differently from the synchronous case in Section 4.3, we do not use an external schedule relation on locations. Instead, nondeterminism appears spontaneously. Indeed, we can implement something similar to Lafont's example [61, B.1]. Suppose $\vdash t\colon\varphi$ and $\vdash u\colon\varphi$ are both derivable. Then, $\vdash (*_{\leftarrow c}^{\rightarrow \bar{c}})t \parallel (*_{\leftarrow \bar{c}}^{\rightarrow c})u\colon\varphi$ is derivable. From there $[((*_{\leftarrow c}^{\rightarrow \bar{c}})t \parallel (*_{\leftarrow \bar{c}}^{\rightarrow c})u), \pi], S_\epsilon$ can reduce both to $[t, \pi], S_\epsilon$ and to $[u, \pi], S_\epsilon$ (Figure 4.3).

4.2.2 Statics

For a set S, form(S) is the set of *S-formulae* φ:

$$\varphi ::= s \mid X \mid \varphi \multimap \varphi \mid \varphi \oplus \varphi \mid \forall X \varphi$$

$$
\begin{array}{ll}
\quad [((\ast^{\to\bar{c}}_{\leftarrow c})t \parallel (\ast^{\to c}_{\leftarrow\bar{c}})u),\pi], S_\epsilon & \quad [((\ast^{\to\bar{c}}_{\leftarrow c})t \parallel (\ast^{\to c}_{\leftarrow\bar{c}})u),\pi], S_\epsilon \\
\succ_{\mathsf{a}} [(\ast^{\to\bar{c}}_{\leftarrow c})t,\pi \parallel (\ast^{\to c}_{\leftarrow\bar{c}})u,\pi], S_\epsilon & \succ_{\mathsf{a}} [(\ast^{\to\bar{c}}_{\leftarrow c})t,\pi \parallel (\ast^{\to c}_{\leftarrow\bar{c}})u,\pi], S_\epsilon \\
\succ_{\mathsf{a}} [\ast_{\leftarrow c},\pi \parallel (\ast^{\to c}_{\leftarrow\bar{c}})u,\pi], S_\epsilon[d\mapsto t] & \succ_{\mathsf{a}} [(\ast^{\to\bar{c}}_{\leftarrow c})t,\pi \parallel \ast_{\leftarrow\bar{c}},\pi], S_\epsilon[c\mapsto u] \\
\succ_{\mathsf{a}} [(\ast^{\to c}_{\leftarrow\bar{c}})u,\pi], S_\epsilon[c\mapsto\top,\bar{c}\mapsto t] & \succ_{\mathsf{a}} [(\ast^{\to\bar{c}}_{\leftarrow c})t,\pi], S_\epsilon[c\mapsto u,\bar{c}\mapsto\top] \\
\succ_{\mathsf{a}} [\ast_{\leftarrow\bar{c}},\pi], S_\epsilon[c\mapsto\bot,\bar{c}\mapsto t] & \succ_{\mathsf{a}} [\ast_{\leftarrow c},\pi], S_\epsilon[c\mapsto u,\bar{c}\mapsto\bot] \\
\succ_{\mathsf{a}} [t,\pi], S_\epsilon[c\mapsto\bot,\bar{c}\mapsto\bot] \ . & \succ_{\mathsf{a}} [u,\pi], S_\epsilon[c\mapsto\bot,\bar{c}\mapsto\bot] \ .
\end{array}
$$

Figure 4.3: A non-confluent example similar to Lafont's example [61, B.1]. Both sides start from the same configuration but reduce to different configurations. Moreover, since terms t and u can be taken arbitrarily, if we equate the terms related by $\succ_{\mathsf{a}}$, we have to conclude that arbitrary two terms of the same type are equal.

where $s \in S$ and $X \in \mathsf{PVar}$. The X in the last clause is binding. The connective $\forall$ connects stronger than $\oplus$, which is stronger than $\multimap$. Repeated $\multimap$'s, $\varphi_0 \multimap \varphi_1 \multimap \cdots \multimap \varphi_n$, are defined inductively on n as $\varphi_0 \multimap (\varphi_1 \multimap \cdots \multimap \varphi_n)$. A *type* is an element of form$(\emptyset)$.

A *sequent* $\Gamma \vdash t\colon\varphi$ consists of a context Γ and an s-term t associated with a type φ. The *context* is a finite sequence of variables associated with types where the same variable does not appear more than once. When we write a concatenation of contexts Γ, Δ, we assume no variables appear in both Γ and Δ. A *hypersequent* is a finite sequence of sequents. In a hypersequent, we also assume that no variable appears more than once in the contexts.

An $\mathbf{N_{MTL2}}$ *derivation* is a tree composed of the derivation rules in Figure 4.4 whose top rules are Ax. A hypersequent is *derivable* in $\mathbf{N_{MTL2}}$ when there is an $\mathbf{N_{MTL2}}$ derivation ending at the hypersequent. An s-term t has type φ when $\vdash t\colon\varphi$ is derivable. A type φ is provable when there is an s-term t with type φ. There is a well-known substructural logic called the monoidal t-norm logic (MTL), which validates the prelinearity axiom $(\varphi \multimap \psi) \oplus (\psi \multimap \varphi)$. Actually, $\mathbf{N_{MTL2}}$ characterizes MTL.

For example, there is an s-term typed with the prelinearity axiom (Figure 4.5).

Corollary 4.2.1 (Prelinearity as a communication scheme) *Assume that an a-term g has type $\forall X \forall Y((X \multimap Y) \oplus (Y \multimap X))$. Then, the s-executable*

$$[g, \langle \mathsf{inl}(z).(z, x\cdot\pi_Y)/\mathsf{inr}(w).(w, y\cdot\pi_X)\rangle]$$

reduces to a multiset containing an element of $\{(x,\pi_X),(y,\pi_Y)\}$.

$$\text{EW}\;\frac{\mathcal{H}}{\mathcal{H} \mid \Gamma \vdash \mathsf{abort} : \varphi} \qquad \text{EE}\;\frac{\mathcal{H} \mid \Gamma \vdash t : \varphi \mid \Delta \vdash u : \psi \mid \mathcal{H}'}{\mathcal{H} \mid \Delta \vdash u : \psi \mid \Gamma \vdash t : \varphi \mid \mathcal{H}'}$$

$$\text{EC}\;\frac{\mathcal{H} \mid \vdash t : \varphi \mid \vdash u : \varphi}{\mathcal{H} \mid \vdash (t \parallel u) : \varphi}$$

$$\text{Ax}\;\frac{}{x : \varphi \vdash x : \varphi} \qquad \text{IE}\;\frac{\mathcal{H} \mid \Gamma, x : \varphi, y : \psi, \Delta \vdash t : \theta}{\mathcal{H} \mid \Gamma, y : \psi, x : \varphi, \Delta \vdash t : \theta} \qquad \text{IW}\;\frac{\mathcal{H} \mid \Gamma \vdash t : \varphi}{\mathcal{H} \mid x : \psi, \Gamma \vdash t : \varphi}$$

$$\multimap\text{I}\;\frac{\mathcal{H} \mid x : \varphi, \Gamma \vdash t : \psi}{\mathcal{H} \mid \Gamma \vdash \lambda x.t : \varphi \multimap \psi} \qquad \multimap\text{E}\;\frac{\mathcal{H} \mid \Gamma \vdash t : \varphi \multimap \psi \qquad \mathcal{H}' \mid \Delta \vdash u : \varphi}{\mathcal{H} \mid \mathcal{H}' \mid \Gamma, \Delta \vdash (t)u : \psi}$$

$$\oplus\text{I}\;\frac{\mathcal{H} \mid \Gamma \vdash t : \varphi}{\mathcal{H} \mid \Gamma \vdash \mathsf{inl}(t) : \varphi \oplus \psi} \qquad \oplus\text{I}\;\frac{\mathcal{H} \mid \Gamma \vdash t : \psi}{\mathcal{H} \mid \Gamma \vdash \mathsf{inr}(t) : \varphi \oplus \psi}$$

$$\oplus\text{E}\;\frac{\mathcal{H} \mid \Gamma \vdash t : \varphi \oplus \psi \qquad \mathcal{H}' \mid \Delta, x : \varphi \vdash u_0 : \theta \qquad \mathcal{H}' \mid \Delta, y : \psi \vdash u_1 : \theta}{\mathcal{H} \mid \mathcal{H}' \mid \Gamma, \Delta \vdash \mathsf{match}\, t\, \mathsf{of}\ \mathsf{inl}(x).u_0/\mathsf{inr}(y).u_1 : \theta}$$

$$\forall\text{I}\;\frac{\mathcal{H} \mid \Gamma \vdash t : \varphi}{\mathcal{H} \mid \Gamma \vdash t : \forall X \varphi}\ (\text{no free } X \text{ in } \mathcal{H} \text{ or } \Gamma) \qquad \forall\text{E}\;\frac{\mathcal{H} \mid \Gamma \vdash t : \forall X \varphi}{\mathcal{H} \mid \Gamma \vdash t : \varphi[\psi/X]}$$

$$\text{Com}\;\frac{\mathcal{H} \mid x : \varphi \multimap \psi, \Gamma \vdash t : \theta \qquad \mathcal{H}' \mid y : \psi \multimap \varphi, \Delta \vdash u : \tau}{\mathcal{H} \mid \mathcal{H}' \mid \Gamma \vdash t[*^{\rightarrow \bar{c}}_{\leftarrow c}/x] : \theta \mid \Delta \vdash u[*^{\rightarrow c}_{\leftarrow \bar{c}}/y] : \tau}$$

Figure 4.4: Typing derivation rules of $\mathbf{N_{MTL2}}$. $\mathcal{H}$ and $\mathcal{H}'$ stand for hypersequents. In EW, $\mathcal{H}$ cannot be empty.

$$\forall\text{I}\;\dfrac{\forall\text{I}\;\dfrac{\text{EC}\;\dfrac{\oplus\text{I}\;\dfrac{\oplus\text{I}\;\dfrac{\text{Com}\;\dfrac{\text{Ax}\;\dfrac{}{x : X \multimap Y \vdash x : X \multimap Y} \qquad \text{Ax}\;\dfrac{}{y : Y \multimap X \vdash y : Y \multimap X}}{\vdash *^{\rightarrow \bar{c}}_{\leftarrow c} : X \multimap Y \mid \vdash *^{\rightarrow c}_{\leftarrow \bar{c}} : Y \multimap X}}{\vdash \mathsf{inl}(*^{\rightarrow \bar{c}}_{\leftarrow c}) : (X \multimap Y) \oplus (Y \multimap X) \mid \vdash *^{\rightarrow c}_{\leftarrow \bar{c}} : Y \multimap X}}{\vdash \mathsf{inl}(*^{\rightarrow \bar{c}}_{\leftarrow c}) : (X \multimap Y) \oplus (Y \multimap X) \mid \vdash \mathsf{inr}(*^{\rightarrow c}_{\leftarrow \bar{c}}) : (X \multimap Y) \oplus (Y \multimap X)}}{\vdash (\mathsf{inl}(*^{\rightarrow \bar{c}}_{\leftarrow c}) \parallel \mathsf{inr}(*^{\rightarrow c}_{\leftarrow \bar{c}})) : (X \multimap Y) \oplus (Y \multimap X)}}{\vdash (\mathsf{inl}(*^{\rightarrow \bar{c}}_{\leftarrow c}) \parallel \mathsf{inr}(*^{\rightarrow c}_{\leftarrow \bar{c}})) : \forall Y((X \multimap Y) \oplus (Y \multimap X))}}{\vdash (\mathsf{inl}(*^{\rightarrow \bar{c}}_{\leftarrow c}) \parallel \mathsf{inr}(*^{\rightarrow c}_{\leftarrow \bar{c}})) : \forall X \forall Y((X \multimap Y) \oplus (Y \multimap X))}$$

Figure 4.5: A derivation tree typing an s-term with the prelinearity axiom: the s-term $(\mathsf{inl}(*^{\rightarrow \bar{c}}_{\leftarrow c}) \parallel \mathsf{inr}(*^{\rightarrow c}_{\leftarrow \bar{c}}))$ has type $\forall X \forall Y((X \multimap Y) \oplus (Y \multimap X))$.

PROOF By Props. 4.3.5 and 4.3.6 below. ■

4.3 Synchronous Semantics

We consider a programming language, which is a modification of Danos and Krivine [38]'s. We assume a set PVar of propositional variables whose cardinality is countably infinite. We also assume countably infinitely many locations with involution satisfying $\bar{c} \neq c$ and $\bar{\bar{c}} = c$. Metavariable c runs over locations. An *s-term* (synchronous term) t is defined by a BNF:

$$t ::= x \mid (t)t \mid \lambda x.t \mid \mathsf{match}\, t\, \mathsf{of}\, \mathsf{inl}(x).t/\mathsf{inr}(x).t \mid \mathsf{inl}(t) \mid \mathsf{inr}(t) \mid$$
$$(t \parallel t) \mid *^{\to c}_{\leftarrow c} \mid \mathsf{abort}$$

where x is a variable and c is a location. The variable occurrences except in the first clause are binding. An *s-stack* (synchronous stack) π is defined by a BNF:

$$\pi ::= \epsilon \mid t \cdot \pi \mid \langle \mathsf{inl}(x).(t,\pi)/\mathsf{inr}(x).(t,\pi)\rangle \ .$$

We write the set of s-terms as Λ_{s} and s-stacks Π_{s}. For s-terms s, t and a variable x, $s[t/x]$ denotes the result of substitution of t for free occurrences of x in s. When more than one substitutions are concatenated, e.g. $s[t/x][t'/y]$, the substitutions are applied at the same time not one after another. When we use this kind of simultaneous substitution, we always make sure that the same variable x does not appear more than once after the /'s, e.g. $s[t/x][t'/x]$ never happens.

A *process* is an element of $\Lambda_{\mathsf{s}} \times \Pi_{\mathsf{s}}$. An *s-executable* (synchronous executable) is a multiset of processes. We ruthlessly use $\parallel$ both for delimiting elements in an s-executable and for addition of multisets. We denote the empty multiset by $\emptyset$.

A *schedule* is a total preorder on locations. Given a fixed schedule $\sqsubseteq$, we define binary relations $\succ'_{\mathsf{s}}$ and $\succ_{\mathsf{s}}$ on s-executables to be the smallest preorders that satisfy:

(cong) if $e_0 \succ'_{\mathsf{s}} e_1$ then $e_0 \parallel e \succ'_{\mathsf{s}} e_1 \parallel e$;

(abort) $[\mathsf{abort}, \pi] \succ'_{\mathsf{s}} \emptyset$;

(push) $[(t)u, \pi] \succ'_{\mathsf{s}} [t, u \cdot \pi]$;

(store) $[\lambda x.t, u \cdot \pi] \succ'_{\mathsf{s}} [t[u/x], \pi]$;

(ask) $[\mathsf{match}\, t\, \mathsf{of}\, \mathsf{inl}(x).u/\mathsf{inr}(y).v, \pi] \succ'_{\mathsf{s}} [t, \langle \mathsf{inl}(x).(u,\pi)/\mathsf{inr}(y).(v,\pi)\rangle]$;

(ansL) $[\mathsf{inl}(v), \langle \mathsf{inl}(x).(t,\pi)/\mathsf{inr}(y).(u,\sigma)\rangle] \succ'_{\mathsf{s}} [t[v/x], \pi]$;

(ansR) $[\mathsf{inr}(w), \langle \mathsf{inl}(x).(t,\pi)/\mathsf{inr}(y).(u,\sigma)\rangle] \succ'_{\mathsf{s}} [u[w/y],\sigma]$;

(com0) $[*^{\rightarrow \bar{c}}_{\leftarrow c}, t \cdot \pi \parallel *^{\rightarrow c}_{\leftarrow \bar{c}}, u \cdot \sigma] \succ'_{\mathsf{s}} [u,\pi]$ (if $\bar{c} \sqsubseteq c$ but $c \not\sqsubseteq \bar{c}$) ;

(com1) $[*^{\rightarrow \bar{c}}_{\leftarrow c}, t \cdot \pi \parallel *^{\rightarrow c}_{\leftarrow \bar{c}}, u \cdot \sigma] \succ'_{\mathsf{s}} [t,\sigma]$ (if $c \sqsubseteq \bar{c}$ but $\bar{c} \not\sqsubseteq c$) ;

(com2) $[*^{\rightarrow \bar{c}}_{\leftarrow c}, t \cdot \pi \parallel *^{\rightarrow c}_{\leftarrow \bar{c}}, u \cdot \sigma] \succ'_{\mathsf{s}} [u,\pi \parallel t,\sigma]$ (if $c \sqsubseteq \bar{c}$ and $\bar{c} \sqsubseteq c$) ;

(local-global) If $e_0 \succ'_{\mathsf{s}} e_1$ then $e_0 \succ_{\mathsf{s}} e_1$; and

(dist) $[(t \parallel u),\pi \parallel e] \succ_{\mathsf{s}} [(t,\pi) \parallel e \parallel (u,\pi) \parallel e]$.

We say e reduces to e' when $e \succ_{\mathsf{s}} e'$ holds. Below, we sometimes omit the outermost parentheses (i.e. [and]) for multisets. Rules (cong), (push) and (store) come from Danos and Krivine [38]. (dist) also appears there but we changed it so that the proof for adequacy (Theorem 4.3.3) goes through in the case (EC, −).

Definition 4.3.1 *A pole* $\perp\!\!\!\perp$ *is a set of s-executables which satisfies*

1. *e is in $\perp\!\!\!\perp$ if $e \succ_{\mathsf{s}} e'$ and $e' \in \perp\!\!\!\perp$; and*
2. *$e \parallel f$ is in $\perp\!\!\!\perp$ if e or f is in $\perp\!\!\!\perp$.*

In the definition of poles, condition 2. is different from that of Danos and Krivine's [38]. There, the condition says if e *and* f *are* in $\perp\!\!\!\perp$, then $e \parallel f$ is in $\perp\!\!\!\perp$. Our disjunctive choice here is influenced by hypersequents [8] and hyper-lambda calculi (Chapter 3), where components are interpreted disjunctively. Computationally, we only guarantee that at least one process of an s-executable $[t_0,\pi_0 \parallel \cdots \parallel t_n,\pi_n]$ succeeds.

An *environment* is a pair of an s-stack and an s-executable. The set of environments is written as E. A *program* is a pair of an s-term and an s-executable. For a set $\mathcal{Z}$ of environments, $\mathcal{Z} \to \perp\!\!\!\perp$ denotes the set of programs (t,e) such that for any environment $(\pi,e') \in \mathcal{Z}$, the s-executable $[t,\pi \parallel e \parallel e']$ is in $\perp\!\!\!\perp$. We use programs and environments because if we continued using s-terms and s-stacks, the proof of adequacy (Theorem 4.3.3) would fail in the case ($\multimap$E, −), *(2)*.

For $\varphi \in \mathrm{form}(2^E)$ and $|\cdot|_0^- : \mathsf{PVar} \to 2^E$, we define $|\varphi|^- \in 2^E$ inductively on φ:

$$\begin{aligned}
|\mathcal{Z}|^- =& \mathcal{Z} \text{ for } \mathcal{Z} \in 2^E \\
|X|^- =& |X|_0^- \\
|\varphi \multimap \psi|^- =& \{(t \cdot \pi, e_0 \parallel e_1) \mid (t, e_0) \in |\varphi|^- \to \perp\!\!\!\perp \text{ and } (\pi, e_1) \in |\psi|^-\} \\
|\varphi \oplus \psi|^- =& \{(\langle \mathsf{inl}(x).(t,\pi)/\mathsf{inr}(y).(u,\sigma)\rangle, f) \mid \\
& \quad t[v/x], \pi \parallel f \parallel f' \in \perp\!\!\!\perp \text{ for all } (v, f') \in |\varphi|^- \to \perp\!\!\!\perp \text{ and} \\
& \quad u[w/y], \sigma \parallel f \parallel f' \in \perp\!\!\!\perp \text{ for all } (w, f') \in |\psi|^- \to \perp\!\!\!\perp\} \\
|\forall X \varphi|^- =& \bigcup_{\mathcal{Z} \in 2^{\Pi}_{\mathsf{s}}} |\varphi[\mathcal{Z}/X]|^- \ .
\end{aligned}$$

Using this, we define $|\varphi| = |\varphi|^- \to \perp\!\!\!\perp$. We have an equality $|\varphi \multimap \psi|^- = \{(t \cdot \pi, e_0 \parallel e_1) \mid (t, e_0) \in |\varphi| \text{ and } (\pi, e_1) \in |\psi|^-\}$. Moreover, for types φ and ψ, we define $|(\varphi, \psi)|$ as the set of triples (t, u, e) of s-terms t and u and an s-executable e such that $[t, \pi \parallel u, \sigma \parallel e \parallel e_0 \parallel e_1] \in \perp\!\!\!\perp$ for any $(\pi, e_0) \in |\varphi|^-$ and $(\sigma, e_1) \in |\psi|^-$.

Proposition 4.3.2 $(*^{\rightarrow \bar{c}}_{\leftarrow c}, *^{\rightarrow c}_{\leftarrow \bar{c}}, \emptyset) \in |(\varphi \multimap \psi, \psi \multimap \varphi)|$ *for any types* φ *and* ψ.

Proof Take any $(t \cdot \sigma, e_0 \parallel e_1) \in |\varphi \multimap \psi|^-$ and $(u \cdot \pi, f_0 \parallel f_1) \in |\psi \multimap \varphi|^-$ such that $(t, e_0) \in |\varphi|$, $(\sigma, e_1) \in |\psi|^-$, $(u, f_0) \in |\psi|$ and $(\pi, f_1) \in |\varphi|^-$ hold. We claim that $e = [*^{\rightarrow \bar{c}}_{\leftarrow c}, t \cdot \sigma \parallel *^{\rightarrow c}_{\leftarrow \bar{c}}, u \cdot \pi \parallel e_0 \parallel e_1 \parallel f_0 \parallel f_1]$ is in $\perp\!\!\!\perp$. Depending on the schedule, e might reduce to $[t, \pi \parallel e_0 \parallel e_1 \parallel f_0 \parallel f_1]$, $[u, \sigma \parallel e_0 \parallel e_1 \parallel f_0 \parallel f_1]$ or $[t, \pi \parallel u, \sigma \parallel e_0 \parallel e_1 \parallel f_0 \parallel f_1]$, all of which are in $\perp\!\!\!\perp$ by condition 2. of Definition 4.3.1 because $[t, \pi \parallel e_0 \parallel f_1]$ and $[u, \sigma \parallel e_1 \parallel f_0]$ are in $\perp\!\!\!\perp$. Since $\perp\!\!\!\perp$ is closed for $\succ_{\mathsf{s}}^{-1}$, we have $e \in \perp\!\!\!\perp$. ■

For $\Gamma = x_1 : \varphi_1, \ldots, x_n : \varphi_n$, we denote by $|\Gamma|$ the set of pairs $(\overrightarrow{t}, e)$ where $\overrightarrow{t} = (t_1, \ldots, t_n)$, $e = ||_{1 \leq i \leq n}\, e_i$ and each pair (t_i, e_i) is in $|\varphi_i|$. For that $\overrightarrow{t}$, $[\overrightarrow{t}/\Gamma]$ denotes the simultaneous substitution $[t_i/x_i]_{0 \leq i \leq n}$. For a hypersequent, we define a set of s-executables $[\![\Gamma_0 \vdash t_0 : \varphi_0 \ \rule{2pt}{8pt}\ \cdots \ \rule{2pt}{8pt}\ \Gamma_n \vdash t_n : \varphi_n]\!]$ to be the set of executables of the form $||_{0 \leq i \leq n}\, (t_i[\overrightarrow{g_i}/\Gamma_i], \pi_i \parallel e_i \parallel f_i)$ where $(\overrightarrow{g_i}, e_i) \in |\Gamma_i|$ and $(\pi_i, f_i) \in |\varphi_i|^-$.

Here we state adequacy. What we will use later is statement *(1)*. However, when we try to prove *(1)* by induction on derivations, the case for Com rule cannot be proved due to insufficient induction hypotheses. Thus we deal with two derivations at the same time.

Theorem 4.3.3 (Adequacy of NMTL2) *Let hypersequents* $\mathcal{H}$ *and* $\mathcal{I}$ *be derivable. We state:*

(1) any s-executable in $[\![\mathcal{H}]\!]$ is also in $\perp\!\!\!\perp$; and

(2) when $\mathcal{H}$ and $\mathcal{I}$ are respectively equal to $\hat{\mathcal{H}} \mid x:\theta, \hat{\Gamma} \vdash t:\varphi$ and $\hat{\mathcal{I}} \mid y:\tau, \hat{\Delta} \vdash u:\psi$ up to exchange, the following triple is in $|(\varphi,\psi)|$:

$$\Big(t[v/x][\overrightarrow{g}/\hat{\Gamma}], \quad u[w/y][\overrightarrow{d}/\hat{\Delta}], \quad e \parallel f \parallel e' \parallel e_{\hat{\mathcal{H}}} \parallel e_{\hat{\mathcal{I}}}\Big)$$

given $e_{\hat{\mathcal{H}}} \in [\![\hat{\mathcal{H}}]\!]$, $e_{\hat{\mathcal{I}}} \in [\![\hat{\mathcal{I}}]\!]$, $(v, w, e') \in |(\theta,\tau)|$, $(\overrightarrow{g}, e) \in |\hat{\Gamma}|$ and $(\overrightarrow{d}, f) \in |\hat{\Delta}|$.

Proof We prove both statements at the same time by induction on the sum of the heights of the derivations of $\mathcal{H}$ and $\mathcal{I}$. Here we identify hypersequents up to exchange.

(Ax, Ax) When both derivations consist of only axiom rules, the statements follow from the definitions of $|\varphi|$ and $|(\varphi,\psi)|$.

(IW, —) The derivation for t ends as

$$\mathrm{W}\,\frac{\mathcal{H}' \mid \Gamma' \vdash t:\varphi}{\mathcal{H}' \mid \hat{x}:\hat{\varphi}, \Gamma' \vdash t:\varphi}\ .$$

(1) Take any $e_{\mathcal{H}'} \in [\![\mathcal{H}']\!]$, $(\overrightarrow{g'}, e) \in |\Gamma'|$ and $(\hat{t}, \hat{e}) \in |\hat{\varphi}|$. Since $\hat{x}$ does not appear freely in t, the term in question $t[\hat{t}/\hat{x}][\overrightarrow{g'}/\Gamma']$ is equal to $t[\overrightarrow{g'}/\Gamma']$. By the induction hypothesis *(1)*, $(t[\overrightarrow{g'}/\Gamma'], e \parallel e_{\mathcal{H}'})$ is in $|\varphi|$. By condition 2. of Definition 4.3.1, the program in question $(t[\hat{t}/\hat{x}][\overrightarrow{g'}/\Gamma'], e \parallel \hat{e} \parallel e_{\mathcal{H}'})$ is also in $|\varphi|$.

(2) When x and $\hat{x}$ are different, we can do the same as for *(1)*. Otherwise, x is equal to $\hat{x}$ and θ is equal to $\hat{\varphi}$. Take any $e_{\hat{\mathcal{H}}} \in [\![\hat{\mathcal{H}}]\!]$, $e_{\hat{\mathcal{I}}} \in [\![\hat{\mathcal{I}}]\!]$, $(v, w, e') \in |(\theta,\tau)|$, $(\overrightarrow{g}, e) \in |\hat{\Gamma}|$, $(\overrightarrow{d}, f) \in |\hat{\Delta}|$, $(\pi, e'') \in |\varphi|^-$ and $(\sigma, f'') \in |\psi|^-$. We have to show that the following executable is in $\perp\!\!\!\perp$:

$$\bar{e} = t[v/\hat{x}][\overrightarrow{g}/\hat{\Gamma}], \pi \parallel u[w/y][\overrightarrow{d}/\hat{\Delta}], \sigma \parallel e \parallel f \parallel e' \parallel e'' \parallel f'' \parallel e_{\hat{\mathcal{H}}} \parallel e_{\hat{\mathcal{I}}}\ .$$

By the induction hypothesis *(1)*, $[t[\overrightarrow{g}/\hat{\Gamma}], \pi \parallel e \parallel e'' \parallel e_{\hat{\mathcal{H}}}]$ is in $\perp\!\!\!\perp$. Moreover, since $\hat{x}$ does not appear in t, $[t[v/\hat{x}][\overrightarrow{g}/\hat{\Gamma}], \pi \parallel e \parallel e'' \parallel e_{\hat{\mathcal{H}}}]$ is in $\perp\!\!\!\perp$. By condition 2. of Definition 4.3.1, $\bar{e}$ is also in $\perp\!\!\!\perp$.

(EW, —) *(1)* By (abort) reduction.

(2) Also by (abort) reduction, but when x appears in the context for abort, we have to do the same argument as in (W, —), *(2)*.

(EC, —) The derivation for t ends as

$$\frac{\mathcal{H}' \mid \ \vdash t_0:\varphi \mid \ \vdash t_1:\varphi}{\mathcal{H}' \mid \ \vdash (t_0 \parallel t_1):\varphi}\ .$$

(1) Take any $e_{\mathcal{H}'} \in [\![\mathcal{H}']\!]$ and $(\pi, f) \in |\varphi|^-$. We have to show that this executable is in $⫫$:

$$[e_{\mathcal{H}'} \parallel (t_0 \parallel t_1), \pi \parallel f] \ .$$

By (dist), this executable reduces to

$$[e_{\mathcal{H}'} \parallel e_{\mathcal{H}'} \parallel t_0, \pi \parallel t_1, \pi \parallel f \parallel f] \ ,$$

which is in $⫫$ because, by the induction hypothesis, the following executable is in $⫫$:

$$[e_{\mathcal{H}'} \parallel (t_0 \parallel t_1), \pi \parallel f \parallel f] \ .$$

(2) Similar to *(1)*.

($\multimap$I, $-$) The derivation for t ends as

$$\frac{\mathcal{H}' \ \rule{2pt}{1em}\ \hat{x}\!:\!\varphi_0, \Gamma \vdash t_1\!:\!\varphi_1}{\mathcal{H}' \ \rule{2pt}{1em}\ \Gamma \vdash \lambda\hat{x}.t_1\!:\!\varphi_0 \multimap \varphi_1} \ .$$

(1) Take any $e_{\mathcal{H}'} \in [\![\mathcal{H}']\!]$, $(\overrightarrow{g}, e) \in |\Gamma|$ and $(t_0 \cdot \pi, e_0 \parallel e_1) \in |\varphi_0 \multimap \varphi_1|^-$ so that $(t_0, e_0) \in |\varphi_0|$ and $(\pi, e_1) \in |\varphi_1|^-$ hold. We have to show that the following s-executable is in $⫫$:

$$(\lambda\hat{x}.t_1)[\overrightarrow{g}/\Gamma], t_0 \cdot \pi \parallel e \parallel e_0 \parallel e_1 \parallel e_{\mathcal{H}'} \ .$$

By (store) and (cong), the s-executable reduces to

$$t_1[t_0/\hat{x}][\overrightarrow{g}/\Gamma], \pi \parallel e \parallel e_0 \parallel e_1 \parallel e_{\mathcal{H}'} \ .$$

We have $(t_0, e_0) \in |\varphi_0|$ so, by the induction hypothesis *(1)*, the reduct is in $⫫$. Since $⫫$ is closed for ${\succ_{\mathsf{s}}}^{-1}$, the original s-executable is in $⫫$, too.

(2) Take any $e_{\hat{\mathcal{H}}} \in [\![\hat{\mathcal{H}}]\!]$, $e_{\hat{\mathcal{I}}} \in [\![\hat{\mathcal{I}}]\!]$, $(v, w, e') \in |(\theta, \tau)|$, $(\overrightarrow{g}, e) \in |\hat{\Gamma}|$, $(\overrightarrow{d}, f) \in |\hat{\Delta}|$, $(t_0 \cdot \pi, e'_0 \parallel e'_1) \in |\varphi_0 \multimap \varphi_1|^-$ and $(\sigma, f'') \in |\psi|^-$ so that $(t_0, e'_0) \in |\varphi_0|$ and $(\pi, e'_1) \in |\varphi_1|^-$. We have to show that this s-executable is in $⫫$:

$$(\lambda\hat{x}.t_1)[v/x][\overrightarrow{g}/\Gamma], t_0\pi \parallel u[w/y][\overrightarrow{d}/\Delta], \sigma \parallel e' \parallel e \parallel f \parallel e'_0 \parallel e'_1 \parallel e_{\hat{\mathcal{H}}} \parallel e_{\hat{\mathcal{I}}} \ .$$

By (store) and (cong), the s-executable reduces to

$$t_1[v/x][\overrightarrow{g}/\Gamma][t_0/\hat{x}], \pi \parallel u[w/y][\overrightarrow{d}/\Delta], \sigma \parallel e' \parallel e \parallel f \parallel e'_0 \parallel e'_1 \parallel e_{\hat{\mathcal{H}}} \parallel e_{\hat{\mathcal{I}}} \ .$$

Since (t_0, e'_0) is in $|\varphi_0|$, by the induction hypothesis *(2)*, the reduct is in $⫫$. Since $⫫$ is closed for ${\succ_{\mathsf{s}}}^{-1}$, the original s-executable is also in $⫫$.

($\multimap$E, $-$) The derivation for t ends as

$$\frac{\mathcal{H}_0 \mathrel{\rule{1pt}{1em}} \Gamma_0 \vdash t_0 : \varphi' \multimap \varphi \qquad \mathcal{H}_1 \mathrel{\rule{1pt}{1em}} \Gamma_1 \vdash t_1 : \varphi'}{\mathcal{H}_0 \mathrel{\rule{1pt}{1em}} \mathcal{H}_1 \mathrel{\rule{1pt}{1em}} \Gamma_0, \Gamma_1 \vdash (t_0)t_1 : \varphi} \; .$$

(1) Take any $e_{\mathrm{H}0} \in [\![\mathcal{H}_0]\!]$, $e_{\mathrm{H}1} \in [\![\mathcal{H}_1]\!]$, $(\overrightarrow{g_0}, e_0) \in |\Gamma_0|$, $(\overrightarrow{g_1}, e_1) \in |\Gamma_1|$ and $(\pi, e'') \in |\varphi|^-$. We have to show that this s-executable is in $\perp\!\!\!\perp$:

$$((t_0)t_1)[\overrightarrow{g_0}/\Gamma_0][\overrightarrow{g_1}/\Gamma_1], \pi \parallel e_0 \parallel e_1 \parallel e'' \parallel e_{\mathrm{H}0} \parallel e_{\mathrm{H}1}.$$

By (push) and (cong), this s-executable reduces to

$$t_0[\overrightarrow{g_0}/\Gamma_0], t_1[\overrightarrow{g_1}/\Gamma_1] \cdot \pi \parallel e_0 \parallel e_1 \parallel e'' \parallel e_{\mathrm{H}0} \parallel e_{\mathrm{H}1}.$$

By the induction hypothesis *(1)* on both branches, we have $(t_0[\overrightarrow{g_0}/\Gamma_0], e_0 \parallel e_{\mathrm{H}0}) \in |\varphi' \multimap \varphi|$ and $(t_1[\overrightarrow{g_1}/\Gamma_1], e_1 \parallel e_{\mathrm{H}1}) \in |\varphi'|$. By the latter, we have $(t_1[\overrightarrow{g_1}/\Gamma_1] \cdot \pi, e_1 \parallel e'') \in |\varphi' \multimap \varphi|^-$. By definition of $|\varphi' \multimap \varphi|$, we have shown that the reduct is in $\perp\!\!\!\perp$. Since $\perp\!\!\!\perp$ is closed for $\succ_{\mathsf{s}}^{-1}$, the original s-executable is also in $\perp\!\!\!\perp$.

(2) Variable x might be contained in Γ_0, Γ_1, $\mathcal{H}_0$ or $\mathcal{H}_1$. In the last two cases, the proof is similar to that of *(1)*.

(case $\Gamma_0 = x : \theta, \hat{\Gamma}_0$) Take any $e_{\mathrm{H}0} \in [\![\mathcal{H}_0]\!]$, $e_{\mathrm{H}1} \in [\![\mathcal{H}_1]\!]$, $(v, w, e') \in |(\theta, \tau)|$, $(\overrightarrow{g_0}, e_0) \in |\hat{\Gamma}_0|$, $(\overrightarrow{g_1}, e_1) \in |\Gamma_1|$, $(\overrightarrow{d}, f) \in |\hat{\Delta}|$, $(\pi, e'') \in |\varphi|^-$ and $(\sigma, f'') \in |\psi|^-$. We have to show that the following s-executable is in $\perp\!\!\!\perp$:

$$((t_0)t_1)[v/x][\overrightarrow{g_0}/\hat{\Gamma}_0][\overrightarrow{g_1}/\Gamma_1], \pi \parallel u', \sigma \parallel \hat{e} \parallel e_1 \parallel e'' \parallel f'' \parallel e_{\mathrm{H}1} \; .$$

where $u' = u[w/y][\overrightarrow{d}/\hat{\Delta}]$ and $\hat{e} = e' \parallel e_0 \parallel e_{\mathrm{H}0} \parallel f$. By (push) and (cong), this s-executable reduces to

$$t_0[v/x][\overrightarrow{g_0}/\hat{\Gamma}_0], t_1[\overrightarrow{g_1}/\Gamma_1] \cdot \pi \parallel u', \sigma \parallel \hat{e} \parallel e_1 \parallel e'' \parallel f'' \parallel e_{\mathrm{H}1} \; .$$

By the induction hypothesis *(1)*, $(t_1[\overrightarrow{g_1}/\Gamma_1], e_1 \parallel e_{\mathrm{H}1})$ is in $|\varphi'|$. In addition to this, (π, e'') is in $|\varphi|^-$, making $(t_1[\overrightarrow{g_1}/\Gamma_1] \cdot \pi, e_1 \parallel e_{\mathrm{H}1} \parallel e'')$ in $|\varphi' \multimap \varphi|^-$. By the induction hypothesis *(2)*, $(t_0[v/x][\overrightarrow{g_0}/\hat{\Gamma}_0], u', \hat{e})$ is in $|(\varphi' \multimap \varphi, \psi)|$. These make the reduct a member of $\perp\!\!\!\perp$. Since $\perp\!\!\!\perp$ is closed for $\succ_{\mathsf{s}}{}^{-1}$, the original s-executable is also in $\perp\!\!\!\perp$.

(case $\Gamma_1 = x : \theta, \hat{\Gamma}_1$) Take any $e_{\mathrm{H}0} \in [\![\mathcal{H}_0]\!]$, $e_{\mathrm{H}1} \in [\![\mathcal{H}_1]\!]$, $(v, w, e') \in |(\theta, \tau)|$, $(\overrightarrow{g_0}, e_0) \in |\Gamma_0|$, $(\overrightarrow{g_1}, e_1) \in |\hat{\Gamma}_1|$, $(\overrightarrow{d}, f) \in |\hat{\Delta}|$, $(\pi, e'') \in |\varphi|^-$ and

$(\sigma, f'') \in |\psi|^-$. We have to show that the following s-executable is in $\perp\!\!\!\perp$:

$$((t_0)t_1)[v/x][\overrightarrow{g_0}/\Gamma_0][\overrightarrow{g_1}/\hat{\Gamma}_1], \pi \parallel e_0 \parallel e_{\mathrm{H}0} \parallel \hat{f} \parallel e''$$

where $\hat{f} = u[w/y][\overrightarrow{d}/\hat{\Delta}], \sigma \parallel f \parallel f'' \parallel e_1 \parallel e_{\mathrm{H}1} \parallel e'$. By (push) and (cong), this s-executable reduces to

$$t_0[\overrightarrow{g_0}/\Gamma_0], t_1[v/x][\overrightarrow{g_1}/\hat{\Gamma}_1] \cdot \pi \parallel e_0 \parallel e_{\mathrm{H}0} \parallel \hat{f} \parallel e'' \ .$$

By the induction hypothesis *(2)*, $(t_1[v/x][\overrightarrow{g_1}/\hat{\Gamma}_1], u[w/y][\overrightarrow{d}/\hat{\Delta}], e_1 \parallel e_{\mathrm{H}1} \parallel f \parallel e')$ is in $|(\varphi', \psi)|$. So, since (σ, f'') is in $|\psi|^-$, $(t_1[v/x][\overrightarrow{g_1}/\hat{\Gamma}_1], \hat{f})$ is in $|\varphi'|$. Moreover we have $(\pi, e'') \in |\varphi|^-$. Combined, $(t_1[v/x][\overrightarrow{g_1}/\hat{\Gamma}_1]{\cdot}\pi, \hat{f} \parallel e'')$ is in $|\varphi' \multimap \varphi|^-$. By the induction hypothesis *(1)*, $(t_0[\overrightarrow{g_0}/\Gamma_0], \hat{e})$ is in $|\varphi' \multimap \varphi|$. So, the reduct is in $\perp\!\!\!\perp$, making the original s-executable a member of $\perp\!\!\!\perp$.

($\oplus$I, —) Without loss of generality, we assume that the derivation for t ends as

$$\frac{\mathcal{H}' \ \| \ \Gamma \vdash t' : \varphi_0}{\mathcal{H}' \ \| \ \Gamma \vdash \mathsf{inl}(t') : \varphi_0 \oplus \varphi_1} \ .$$

(1) Take any $e_{\mathcal{H}'} \in [\![\mathcal{H}']\!]$, $(\overrightarrow{g}, e) \in |\Gamma|$ and $(\langle \mathsf{inl}(\hat{x}).(v, \pi_0)/\mathsf{inr}(\hat{y}).(w, \pi_1)\rangle, e'')$ in $|\varphi_0 \oplus \varphi_1|^-$. By the induction hypothesis *(1)*, $(t'[\overrightarrow{g}/\Gamma], e \parallel e_{\mathcal{H}'})$ is in $|\varphi_0|$. By definition of $|\varphi_0 \oplus \varphi_1|^-$, the s-executable $[v[t'[\overrightarrow{g}/\Gamma]/\hat{x}], \pi_0 \parallel e \parallel e_{\mathcal{H}'} \parallel e'']$ is in $\perp\!\!\!\perp$. Since $\perp\!\!\!\perp$ is closed for ${\succ_{\mathsf{s}}}^{-1}$, the s-executable

$$\mathsf{inl}(t')[\overrightarrow{g}/\Gamma], \langle \mathsf{inl}(\hat{x}).(v, \pi_0)/\mathsf{inr}(\hat{y}).(w, \pi_1)\rangle \parallel e \parallel e_{\mathcal{H}} \parallel e''$$

is also in $\perp\!\!\!\perp$. This shows the statement because we chose an arbitrary element of $|\varphi_0 \oplus \varphi_1|^-$.

(2) If x is in $\mathcal{H}$, the same argument as *(1)* suffices. Otherwise, take any $e_{\mathcal{H}} \in [\![\mathcal{H}]\!]$ $(v, w, e') \in |(\theta, \tau)|$, $(\overrightarrow{g}, e) \in |\hat{\Gamma}|$, $(\overrightarrow{d}, f) \in |\hat{\Delta}|$,

$$(\langle \mathsf{inl}(\hat{x}).(\hat{v}, \pi_0)/\mathsf{inr}(\hat{y}).(w, \pi_1)\rangle, e'') \in |\varphi_0 \oplus \varphi_1|^-$$

and $(\sigma, f'') \in |\psi|^-$. We have to show that the following s-executable is in $\perp\!\!\!\perp$:

$$(\mathsf{inl}(t))[v/x][\overrightarrow{g}/\hat{\Gamma}], \langle \mathsf{inl}(\hat{x}).(\hat{v}, \pi_0)/\mathsf{inr}(\hat{y}).(w, \pi_1)\rangle \parallel u', \sigma \parallel \hat{e} \parallel e'' \parallel f''$$

where $u' = u[w/y][\overrightarrow{d}/\hat{\Delta}]$ and $\hat{e} = e' \parallel e \parallel\!\parallel e_{\mathcal{H}} \parallel f$. By (ansL), the s-executable reduces to $\hat{v}[t'[v/x][\overrightarrow{g}/\hat{\Gamma}]/\hat{x}], \pi_0 \parallel u', \sigma \parallel \hat{e} \parallel e'' \parallel f''$. By the induction hypothesis *(2)*, $(t'[v/x][\overrightarrow{g}/\hat{\Gamma}], u', \hat{e})$ is in $|(\varphi_0, \psi)|$. So, $(t'[v/x][\overrightarrow{g}/\hat{\Gamma}], [u', \sigma \parallel$

$\hat{e} \parallel f''$]) is in $|\varphi_0|$. By definition of $|\varphi_0 \oplus \varphi_1|^-$, the reduct is in $⫫$. Since $⫫$ is closed for ${\succ_{\mathsf{s}}}^{-1}$, the original s-executable is also in $⫫$.

($\oplus$E, $-$) The derivation for t ends in

$$\frac{\mathcal{H}_0 \mathbin{\blacksquare} \Gamma_0 \vdash t' : \varphi_0 \oplus \varphi_1 \qquad \mathcal{H}_1 \mathbin{\blacksquare} \Gamma_1, \hat{x} : \varphi_0 \vdash t_0 : \varphi \qquad \mathcal{H}_1 \mathbin{\blacksquare} \Gamma_1, \hat{y} : \varphi_1 \vdash t_1 : \varphi}{\mathcal{H}_0 \mathbin{\blacksquare} \mathcal{H}_1 \mathbin{\blacksquare} \Gamma_0, \Gamma_1 \vdash \mathsf{match}\, t' \,\mathsf{of}\, \mathsf{inl}(\hat{x}).t_0 / \mathsf{inr}(\hat{y}).t_1 : \varphi} .$$

(1) Take any $e_{\mathrm{H}0} \in [\![\mathcal{H}_0]\!]$, $e_{\mathrm{H}1} \in [\![\mathcal{H}_1]\!]$, $(\overrightarrow{g_0}, e_0) \in |\Gamma_0|$, $(\overrightarrow{g_1}, e_1) \in |\Gamma_1|$ and $(\pi, e') \in |\varphi|^-$. We have to show that the s-executable

$$(\mathsf{match}\, t' \,\mathsf{of}\, \mathsf{inl}(\hat{x}).t_0/\mathsf{inr}(\hat{y}).t_1)[\overrightarrow{g_0}/\Gamma_0][\overrightarrow{g_1}/\Gamma_1], \pi \parallel e_0 \parallel e_{\mathrm{H}0} \parallel e' \parallel e_1 \parallel e_{\mathrm{H}1}$$

is in $⫫$. By (ask), the s-executable reduces to

$$t'[\overrightarrow{g_0}/\Gamma_0], \langle \mathsf{inl}(\hat{x}).(t_0[\overrightarrow{g_1}/\Gamma_1], \pi)/\mathsf{inr}(\hat{y}).(t_1[\overrightarrow{g_1}/\Gamma_1], \pi)\rangle \parallel e_0 \parallel e_{\mathrm{H}0} \parallel e' \parallel e_1 \parallel e_{\mathrm{H}1} .$$

We claim that $(\langle \mathsf{inl}(\hat{x}).(t_0[\overrightarrow{g_0/\Gamma_0}], \pi)/\mathsf{inr}(\hat{y}).(t_1, \pi)\rangle, e_1 \parallel e_{\mathrm{H}1} \parallel e')$ is in $|\varphi \oplus \psi|^-$ and that $(t'[\overrightarrow{g_0}/\Gamma_0], e_0 \parallel e_{\mathrm{H}0})$ is in $|\varphi \oplus \psi|$. The first claim is shown by the induction hypothesis *(1)* stating that $(t_0[v/\hat{x}][\overrightarrow{g_1}/\Gamma_1], e_1 \parallel e_{\mathrm{H}1} \parallel e'')$ is in $|\varphi|$ for any $(v, e'') \in |\varphi_0|$ and similarly to t_1. The second claim follows from induction hypothesis *(1)* on t'. By the two claims and by the definition of $|\varphi \oplus \psi|$, we have shown that the reduct is in $⫫$ and thence that the original s-executable is in $⫫$.

(2) Variable x might be in Γ_0, Γ_1, $\mathcal{H}_0$ or $\mathcal{H}_1$. In the last two cases, the proof is similar to that of *(1)*.

(case $\Gamma_0 = x : \theta, \hat{\Gamma}_0$) Take any $e_{\mathrm{H}0} \in [\![\mathcal{H}_0]\!]$, $e_{\mathrm{H}1} \in [\![\mathcal{H}_1]\!]$, $(v, w, e') \in |(\theta, \tau)|$, $(\overrightarrow{g_0}, e_0) \in |\hat{\Gamma}_0|$, $(\overrightarrow{g_1}, e_1) \in |\Gamma_1|$, $(\overrightarrow{d}, f) \in |\hat{\Delta}|$, $(\pi, e'') \in |\varphi|^-$ and $(\sigma, f'') \in |\psi|^-$. We have to show that the following s-executable is in $⫫$:

$$\begin{gathered}(\mathsf{match}\, t' \,\mathsf{of}\, \mathsf{inl}(\hat{x}).t_0/\mathsf{inr}(\hat{y}).t_1)[v/x][\overrightarrow{g_0}/\hat{\Gamma}_0][\overrightarrow{g_1}/\Gamma_1], \pi \parallel \\ u[w/y][\overrightarrow{d}/\hat{\Delta}], \sigma \parallel e' \parallel e_0 \parallel e_{\mathrm{H}0} \parallel e_1 \parallel e_{\mathrm{H}1} \parallel f \parallel e'' \parallel f''.\end{gathered}$$

By (ask) and (cong), this s-executable reduces to

$$\begin{gathered}(t'[v/x][\overrightarrow{g_0}/\hat{\Gamma}_0], \langle \mathsf{inl}(\hat{x}).(t_0[\overrightarrow{g_1}/\Gamma_1], \pi)/\mathsf{inr}(\hat{y}).(t_1[\overrightarrow{g_1}/\Gamma_1], \pi)\rangle \parallel \\ u[w/y][\overrightarrow{d}/\hat{\Delta}], \sigma \parallel e' \parallel e_0 \parallel e_{\mathrm{H}0} \parallel e_1 \parallel e_{\mathrm{H}1} \parallel f \parallel e'' \parallel f'' .\end{gathered}$$

By the induction hypothesis *(2)*, we have

$$(t'[v/x][\overrightarrow{g_0}/\hat{\Gamma}_0], u[w/y][\overrightarrow{d}/\hat{\Delta}], e' \parallel e_0 \parallel e_{\mathrm{H}0} \parallel f) \in |(\varphi_0 \oplus \varphi_1, \psi)| .$$

Also, we defined (σ, f'') to be an element of $|\psi|^-$. By definition of $|(\varphi_0 \oplus \varphi_1, \psi)|$, the reduct is in $⫫$. Since $⫫$ is closed for ${\succ_\mathsf{s}}^{-1}$, the original s-executable is also in $⫫$.

(case $\Gamma_1 = x\!:\!\theta, \hat{\Gamma}_1$) Take any $e_{\mathrm{H}0} \in [\![\mathcal{H}_0]\!]$, $e_{\mathrm{H}1} \in [\![\mathcal{H}_1]\!]$, $(v, w, e') \in |(\theta, \tau)|$, $(\overrightarrow{g_0}, e_0) \in |\Gamma_0|$, $(\overrightarrow{g_1}, e_1) \in |\hat{\Gamma}_1|$, $(\overrightarrow{d}, f) \in |\hat{\Delta}|$, $(\pi, e'') \in |\varphi|^-$ and $(\sigma, f'') \in |\psi|^-$. We have to show that the following s-executable is in $⫫$:

$$(\mathsf{match}\, t' \,\mathsf{of}\, \mathsf{inl}(\hat{x}).t_0/\mathsf{inr}(\hat{y}).t_1)[\overrightarrow{g_0}/\Gamma_0][v/x][\overrightarrow{g_1}/\Gamma_1], \pi \parallel$$
$$u[w/y][\overrightarrow{d}/\hat{\Delta}], \sigma \parallel e' \parallel e_0 \parallel e_{\mathrm{H}0} \parallel e_1 \parallel e_{\mathrm{H}1} \parallel f \parallel e'' \parallel f'' \ .$$

By (ask) and (cong), this s-executable reduces to

$$e_{\mathrm{r}} = t'[\overrightarrow{g_0}/\Gamma_0], \langle \mathsf{inl}(\hat{x}).(t'_0, \pi)/\mathsf{inr}(\hat{y}).(t'_1, \pi)\rangle \parallel$$
$$u[w/y][\overrightarrow{d}/\hat{\Delta}], \sigma \parallel e' \parallel e_0 \parallel e_{\mathrm{H}0} \parallel e_1 \parallel e_{\mathrm{H}1} \parallel f \parallel e'' \parallel f''$$

where $t'_0 = t_0[v/x][\overrightarrow{g_1}/\Gamma_1]$ and $t'_1 = t'_1[v/x][\overrightarrow{g_1}/\Gamma_1]$. By the induction hypothesis *(2)*, the triple $(t'_0[v'_0/\hat{x}], u[w/y][\overrightarrow{d}/\hat{\Delta}], e' \parallel e_1 \parallel e_{\mathrm{H}1} \parallel e'_0 \parallel f)$ is in $|(\varphi, \psi)|$ for any $(v'_0, e'_0) \in |\varphi_0|$ so that $(t'_0[v'_0/\hat{x}], [u[w/y][\overrightarrow{d}/\hat{D}], \sigma \parallel e' \parallel e_1 \parallel e_{\mathrm{H}1} \parallel e'_0 \parallel f \parallel f''])$ is in $|\varphi|$. We have a symmetric fact for t_1. Thus, the environment

$$(\langle \mathsf{inl}(\hat{x}).(t'_0, \pi)/\mathsf{inr}(\hat{y}).(t'_1, \pi)\rangle,$$
$$[u[w/y][\overrightarrow{d}/\hat{\Delta}], \sigma \parallel e' \parallel e_1 \parallel e_{\mathrm{H}1} \parallel f \parallel e'' \parallel f''])$$

is in $|\varphi_0 \oplus \varphi_1|^-$. By the induction hypothesis *(1)*, $(t'[\overrightarrow{g_0}/\Gamma_0], e_0 \parallel e_{\mathrm{H}0})$ is in $|\varphi_0 \oplus \varphi_1|$. By definition of $|\varphi_0 \oplus \varphi_1|$, the reduct e_{r} is in $⫫$. Since $⫫$ is closed for ${\succ_\mathsf{s}}^{-1}$, the original s-executable is also in $⫫$.

(Com, —) The derivation for t ends in

$$\frac{\mathcal{H}_0 \ \| \ \hat{x}\!:\!\varphi_0 \multimap \varphi_1, \Gamma_0 \vdash t_0\!:\!\varphi \qquad \mathcal{H}_1 \ \| \ \hat{y}\!:\!\varphi_1 \multimap \varphi_0, \Gamma_1 \vdash t_1\!:\!\varphi'}{\mathcal{H}_0 \ \| \ \mathcal{H}_1 \ \| \ \Gamma_0 \vdash t_0[*^{\rightarrow \bar{c}}_{\leftarrow c}/\hat{x}]\!:\!\varphi \ \| \ \Gamma_1 \vdash t_1[*^{\rightarrow c}_{\leftarrow \bar{c}}/\hat{y}]\!:\!\varphi'} \ .$$

(1) Take any $e_{\mathrm{H}0} \in [\![\mathcal{H}_0]\!]$, $e_{\mathrm{H}1} \in [\![\mathcal{H}_1]\!]$, $(\overrightarrow{g_0}, e_0) \in |\Gamma_0|$ and $(\overrightarrow{g_1}, e_1) \in |\Gamma_1|$. By Proposition 4.3.2, we have $(*^{\rightarrow \bar{c}}_{\leftarrow c}, *^{\rightarrow c}_{\leftarrow \bar{c}}, \emptyset) \in |(\varphi_0 \multimap \varphi_1, \varphi_1 \multimap \varphi_0)|$. By the induction hypothesis *(2)*, the s-terms $t'_0 = t_0[*^{\rightarrow \bar{c}}_{\leftarrow c}/\hat{x}][\overrightarrow{g_0}/\Gamma_0]$ and $t'_1 = t_1[*^{\rightarrow c}_{\leftarrow \bar{c}}/\hat{y}][\overrightarrow{g_1}/\Gamma_1]$ satisfy $(t'_0, t'_1, e_0 \parallel e_{\mathrm{H}0} \parallel e_1 \parallel e_{\mathrm{H}1}) \in |(\varphi, \varphi')|$.

(2) If $x\!:\!\theta$ is in $\mathcal{H}_0$ or $\mathcal{H}_1$, we can apply the same argument as *(1)*. Otherwise, without loss of generality, we can assume $\Gamma_0 = x\!:\!\theta, \hat{\Gamma}_0$ up to exchange.

Take any $(v, w, e') \in |(\theta, \tau)|$, $(\overrightarrow{g_0}, e_0) \in |\hat{\Gamma}_0|$, $(\overrightarrow{g_1}, e_1) \in |\Gamma_1|$, $(\overrightarrow{d}, f) \in |\hat{\Delta}|$, $(\pi, e'') \in |\varphi|^-$, $(\pi', e''') \in |\varphi'|^-$ and $(\sigma, f'') \in |\psi|^-$. After defining $t'_0 = t_0[*{\substack{\rightarrow \bar{c} \\ \leftarrow c}}/\hat{x}][\overrightarrow{g_0}/\hat{\Gamma}_0][v/x]$, $t'_1 = t_1[*{\substack{\rightarrow c \\ \leftarrow \bar{c}}}/\hat{y}][\overrightarrow{g_1}/\Gamma_1]$ and $u' = u[w/y][\overrightarrow{d}/\hat{\Delta}]$, we have to show that this s-executable is in $\perp\!\!\!\perp$:

$$t'_0, \pi \parallel t'_1, \pi' \parallel u', \sigma \parallel e' \parallel e_0 \parallel e_{\mathrm{H0}} \parallel e_1 \parallel e_{\mathrm{H1}} \parallel f \parallel e'' \parallel e''' \parallel f'' \ .$$

Consider the derivation consisting of an axiom $\hat{x} : \varphi_0 \multimap \varphi_1 \vdash \hat{x} : \varphi_0 \multimap \varphi_1$. This derivation is shorter than the derivation for $\mathcal{H}$. We can use the induction hypothesis *(2)* on this derivation and the derivation of u. By this and Proposition 4.3.2, $(*{\substack{\rightarrow \bar{c} \\ \leftarrow c}}, t'_1, e_1 \parallel e_{\mathrm{H1}})$ is in $|(\varphi_0 \multimap \varphi_1, \varphi)|$. So, program $p = (*{\substack{\rightarrow \bar{c} \\ \leftarrow c}}, [t'_1, \pi' \parallel e_1 \parallel e_{\mathrm{H1}} \parallel e'''])$ is in $|\varphi_0 \multimap \varphi_1|$. By the induction hypothesis *(2)* for the derivations for t_0 and u, especially using the program p for substituting $\hat{x}$, we can show that $(t'_0, u', (t'_1, \pi' \parallel e_1 \parallel e_{\mathrm{H1}} \parallel e''' \parallel e_0 \parallel e_{\mathrm{H0}} \parallel e' \parallel f))$ is in $|(\varphi, \psi)|$. Thus, the reduct is in $\perp\!\!\!\perp$. Since $\perp\!\!\!\perp$ is closed for ${\succ_{\mathsf{s}}}^{-1}$, the original s-executable is also in $\perp\!\!\!\perp$.

($\forall$I, —) The derivation for t ends in $\dfrac{\mathcal{H}' \ \big| \ \Gamma \vdash t : \varphi}{\mathcal{H}' \ \big| \ \Gamma \vdash t : \forall X \varphi}$.

(1) Take any $e_{\mathcal{H}'} \in [\![\mathcal{H}']\!]$ and $(\overrightarrow{g}, e) \in |\Gamma|$. Since Γ does not contain X freely, $|\Gamma|$ does not change whatever $|X|_0^-$ is. By induction hypothesis *(1)* for arbitrary $|X|_0^-$, the program $(t[\overrightarrow{g}/\Gamma], e \parallel e_{\mathcal{H}'})$ is in $\bigcap_{\mathcal{Z} \in 2_{\mathsf{s}}^{\Pi}} |\varphi[\mathcal{Z}/X]|$, which is a subset of $\left(\bigcup_{\mathcal{Z} \in 2_{\mathsf{s}}^{\Pi}} |\varphi[\mathcal{Z}/X]|^-\right) \to \perp\!\!\!\perp$.

(2) We can replace X in the derivation of t with a propositional variable X' that does not occur in the derivation of u. We are going to use the induction hypothesis on the renamed derivation with the s-terms and s-stacks taken above. Since θ and $\hat{\Gamma}$ do not contain X freely, we have $(v, w, e') \in |\theta[X'/X], \tau|$, $(\overrightarrow{g}, e \parallel e_{\mathcal{H}}) \in |\hat{\Gamma}[X'/X]|$ and $(\pi, e'') \in |\forall X' \varphi[X'/X]|^-$. By the induction hypothesis *(2)* on the renamed derivation, the s-terms $t' = t[v/x][\overrightarrow{g}/\Gamma]$ and $u' = u[w/y][\overrightarrow{d}/\Delta]$ satisfy $(t', u', e \parallel f \parallel e_{\mathcal{H}}) \in |\varphi[X'/X], \psi'|$ for any $|X'|_0^-$. That is, for $\mathcal{Z} \in 2^E$ that makes $(\pi, e'') \in |\varphi[X'/X][\mathcal{Z}/X']|^-$ and $(\sigma, f'') \in |\psi[\mathcal{Z}/X']|^-$, $t'\pi \parallel u', \sigma \parallel e'' \parallel f''$ is in $\perp\!\!\!\perp$, making $t', \pi \parallel u', \sigma$ an element of $\perp\!\!\!\perp$..

($\forall$E, —) Both statements follow from the induction hypotheses because $|\varphi[\psi/X]|^-$ is a subset of $|\forall X \varphi|^-$.

(Other cases) We can swap t and u and find a symmetric case above. ■

Note that *(1)* uses *(2)* in (Com, $-$) and *(2)* uses *(1)* in ($\multimap$E, $-$) and (IW, $-$) in the proof of Theorem 4.3.3.

Proposition 4.3.4 *Given a set P of processes, the following set is a pole: the set of s-executables that reduces to an s-executable containing an element of P.*

Proposition 4.3.5 (Prelinearity as a communication scheme) *Assume that an s-term g has type $\forall X \forall Y((X \multimap Y) \oplus (Y \multimap X))$. Then, the s-executable*

$$[g, \langle \mathsf{inl}(z).(z, x \cdot \pi_Y)/\mathsf{inr}(w).(w, y \cdot \pi_X)\rangle]$$

reduces to a multiset containing an element of $\{(x, \pi_X), (y, \pi_Y)\}$.

Proof We denote by $\perp\!\!\!\perp$ the set of s-executables that reduce to a multiset containing an element of $\{(x, \pi_X), (y, \pi_Y)\}$. By Proposition 4.3.4, $\perp\!\!\!\perp$ is a pole. Take $|X|^-_0 = \{(\pi_X, \emptyset)\}$ and $|Y|^-_0 = \{(\pi_Y, \emptyset)\}$. We have $x \in |X|$ and $y \in |Y|$. By definition of $|\cdot|^-$, $\langle \mathsf{inl}(z).(z, x \cdot \pi_Y)/\mathsf{inr}(w).(w, y \cdot \pi_X)\rangle$ is in $|(X \multimap Y) \oplus (Y \multimap X)|^-$. By adequacy (Theorem 4.3.3), $[g, \langle \mathsf{inl}(z).(z, x \cdot \pi_Y)/\mathsf{inr}(w).(w, y \cdot \pi_X)\rangle]$ is in $\perp\!\!\!\perp$. ■

Proposition 4.3.6 *Let e be an s-executable. If $e \succ_{\mathsf{s}} e'$ then $e, S_\epsilon \succ_{\mathsf{a}} e', S_\epsilon$.*

Proof Most reduction rules are common. Rules (com0) and (com1) are simulated in Figure 4.3. Rule (com2) can be simulated similarly. ■

This transfers the semantics of prelinearity (Proposition 4.3.5) to the asynchronous case.

Example 4.3.7 (A term performing information exchange.) *Since $g = (\mathsf{inl}(*^{\rightarrow \bar{c}}_{\leftarrow c}) \parallel \mathsf{inr}(*^{\rightarrow c}_{\leftarrow \bar{c}}))$ has the type $\forall X \forall Y((X \multimap Y) \oplus (Y \multimap X))$ (Figure 4.5), g satisfies the condition of Proposition 4.3.5. Indeed, by (dist), $[g, \langle \mathsf{inl}(z).(z, x \cdot \pi_Y)/\mathsf{inr}(w).(w, y \cdot \pi_X)\rangle]$ reduces to*

$$\begin{aligned}&\big[\,\big(\mathsf{inl}(*^{\rightarrow \bar{c}}_{\leftarrow c}), \langle \mathsf{inl}(z).(z, x \cdot \pi_Y)/\mathsf{inr}(w).(w, y \cdot \pi_X)\rangle\big)\\ &\parallel \big(\mathsf{inr}(*^{\rightarrow c}_{\leftarrow \bar{c}}), \langle \mathsf{inl}(z).(z, x \cdot \pi_Y)/\mathsf{inr}(w).(w, y \cdot \pi_X)\rangle\big)\big]\ .\end{aligned}$$

By (ansL), (ansR) and (cong), the s-term above reduces to $[*^{\rightarrow \bar{c}}_{\leftarrow c}, x \cdot \pi_Y \parallel *^{\rightarrow c}_{\leftarrow \bar{c}}, y \cdot \pi_X]$. Depending on the schedule, the s-term above reduces to either $[x, \pi_X]$ or $[y, \pi_Y]$ or $[x, \pi_X \parallel y, \pi_Y]$.

4.4 Characterization of Monoidal t-Norm Logic

There is a well-known substructural logic called monoidal t-norm logic (MTL), which validates the prelinearity axiom $(\varphi \multimap \psi) \oplus (\psi \multimap \varphi)$. Actually, $\mathbf{N}_{\mathbf{MTL2}}$ characterizes MTL (Propositions. 4.4.2 and 4.4.3).

The MTL formulae can be defined as[2]:

$$\varphi ::= \mathbf{0} \mid X \mid \varphi \multimap \psi \mid \varphi \otimes \psi \mid \varphi \oplus \psi \enspace .$$

As an abbreviation, we can introduce $\varphi \wedge \psi = (\varphi \otimes (\varphi \multimap \psi)) \oplus (\psi \otimes (\psi \multimap \varphi))$[3]. We take a hypersequent formulation of MTL by Baaz et al. [13], whose rules we show in Figure 4.6.

$$(\mathrm{ax})\ \frac{}{X \vdash X} \qquad (\mathrm{ax})\ \frac{}{\mathbf{0} \vdash \varphi} \qquad (\mathrm{com})\ \frac{\mathcal{H} \mid \Delta_0, \Gamma_0 \vdash \varphi \qquad \mathcal{H} \mid \Delta_1, \Gamma_1 \vdash \psi}{\mathcal{H} \mid \Delta_0, \Delta_1 \vdash \varphi \mid \Gamma_0, \Gamma_1 \vdash \psi}$$

$$(\mathrm{cut})\ \frac{\mathcal{H} \mid \Gamma \vdash \varphi \qquad \mathcal{H} \mid \varphi, \Delta \vdash \psi}{\mathcal{H} \mid \Gamma, \Delta \vdash \psi} \qquad (\mathrm{w,l})\ \frac{\mathcal{H} \mid \Gamma \vdash \varphi}{\mathcal{H} \mid \Gamma, \psi \vdash \varphi}$$

$$(\mathrm{w,r})\ \frac{\mathcal{H} \mid \Gamma \vdash \mathbf{0}}{\mathcal{H} \mid \Gamma \vdash \varphi} \qquad (\mathrm{EW})\ \frac{\mathcal{H}}{\mathcal{H} \mid \Gamma \vdash \varphi}$$

$$(\otimes, \mathrm{l})\ \frac{\mathcal{H} \mid \Gamma, \varphi, \psi \vdash \chi}{\mathcal{H} \mid \Gamma, \varphi \otimes \psi \vdash \chi} \qquad (\otimes, \mathrm{r})\ \frac{\mathcal{H} \mid \Gamma \vdash \varphi \qquad \mathcal{H} \mid \Delta \vdash \psi}{\mathcal{H} \mid \Gamma, \Delta \vdash \varphi \otimes \psi}$$

$$(\multimap, \mathrm{l})\ \frac{\mathcal{H} \mid \Gamma \vdash \varphi \qquad \mathcal{H} \mid \Delta, \psi \vdash \chi}{\mathcal{H} \mid \Gamma, \Delta, \varphi \multimap \psi \vdash \chi} \qquad (\multimap, \mathrm{r})\ \frac{\mathcal{H} \mid \Gamma, \varphi \vdash \psi}{\mathcal{H} \mid \Gamma \vdash \varphi \multimap \psi}$$

$$(\oplus, \mathrm{l})\ \frac{\mathcal{H} \mid \Gamma, \varphi \vdash \chi \qquad \mathcal{H} \mid \Gamma, \psi \vdash \chi}{\mathcal{H} \mid \Gamma, \varphi \oplus \psi \vdash \chi} \qquad (\oplus, \mathrm{r})\ \frac{\mathcal{H} \mid \Gamma \vdash \varphi_i}{\mathcal{H} \mid \Gamma \vdash \varphi_0 \oplus \varphi_1}$$

Figure 4.6: The propositional rules in $\mathbf{HL}^{\forall}_{\mathbf{BCK}}$ [13], which characterizes monoidal t-norm logic. Axioms are not shown in Baaz et al. [13] so we took them from Ono and Komori [116]. Exchange rules are implicit: the contexts and hypersequents are treated as finite sets rather than sequences. If a formula is provable with the (cut) rule, the formula is also provable without the (cut) rule by Baaz et al. [13, Theorem 3.2].

[2]From the literature [34], the connectives are translated as & into $\otimes$ and $\to$ into $\multimap$.

[3]The translation of $\wedge$ is a tailor-made one for MTL, which is taken from [34, p. 48]. The origin of the translation can be traced back to Cintula et al. [33, Lemma 6.5].

We translate MTL formulae to MTL2 formulae by induction on φ:

$$\begin{aligned}
\mathbf{0}^\bullet &= \forall X X \\
X^\bullet &= X \\
(\varphi \multimap \psi)^\bullet &= \varphi^\bullet \multimap \psi^\bullet \\
(\varphi \otimes \psi)^\bullet &= \forall X((\varphi^\bullet \multimap \psi^\bullet \multimap X) \multimap X) \text{ where } X \text{ does not appear in } \varphi \otimes \psi \\
(\varphi \oplus \psi)^\bullet &= \varphi^\bullet \oplus \psi^\bullet \quad .
\end{aligned}$$

We state that any MTL formula φ is a theorem in $\mathbf{HL}^{\forall}_{\mathbf{BCK}}$ iff $\varphi^\bullet$ is a theorem in $\mathbf{N_{MTL2}}$. In order to reason inductively on hypersequent derivations, we extend the translation $(\cdot)^\bullet$ to sequents and hypersequents. We define $(\Gamma \vdash \varphi)^\bullet$ to be $\Gamma^\bullet \multimap \varphi^\bullet$ where $\Gamma^\bullet$ is the $\otimes$-conjunction of translations of the elements of Γ. We define $\big(\Gamma_0 \vdash \varphi_0 \ \Vert\ \cdots \ \Vert\ \Gamma_n \vdash \varphi_n\big)^\bullet$ to be $(\Gamma_0 \vdash \varphi_0)^\bullet \ \Vert\ \cdots \ \Vert\ (\Gamma_n \vdash \varphi_n)^\bullet$. A substitution instance of an MTL2 formula φ is an MTL formula obtained by substituting an MTL formula for each bound propositional variable in φ.

Proposition 4.4.1 *If a hypersequent is derivable in* $\mathbf{N_{MTL2}}$*, all substitution instances of the hypersequent are derivable in* $\mathbf{HL}^{\forall}_{\mathbf{BCK}}$.

Proof By induction on $\mathbf{N_{MTL2}}$ derivations. ■

Proposition 4.4.2 (Soundness) *When the translation* $\varphi^\bullet$ *is provable in* $\mathbf{N_{MTL2}}$*, the original* φ *is provable in* $\mathbf{HL}^{\forall}_{\mathbf{BCK}}$.

Proof We define an MTL formula φ° inductively on an MTL formula φ so that φ° is a substitution instance of $\varphi^\bullet$ and $\varphi^\circ \multimap \varphi$ is provable in $\mathbf{HL}^{\forall}_{\mathbf{BCK}}$. The definition of φ° follows:

$$\begin{aligned}
\mathbf{0}^\circ &= \mathbf{0} \\
X^\circ &= X \\
(\varphi \multimap \psi)^\circ &= \varphi^\circ \multimap \psi^\circ \\
(\varphi \otimes \psi)^\circ &= (\varphi^\circ \multimap \psi^\circ \multimap (\varphi^\circ \otimes \psi^\circ)) \multimap (\varphi^\circ \otimes \psi^\circ) \\
(\varphi \oplus \psi)^\circ &= \varphi^\circ \oplus \psi^\circ \quad .
\end{aligned}$$

Both claims are immediate. Assume $\varphi^\bullet$ is provable in $\mathbf{N_{MTL2}}$. By the first claim and Proposition 4.4.1, φ° is provable in $\mathbf{HL}^{\forall}_{\mathbf{BCK}}$. By the second claim, φ is provable in $\mathbf{HL}^{\forall}_{\mathbf{BCK}}$. ■

Proposition 4.4.3 (Completeness) *If a hypersequent $\mathcal{H}$ is derivable in* $\mathbf{HL}^{\forall}_{\mathbf{BCK}}$ *without using (cut), then the translation $\mathcal{H}^{\bullet}$ is derivable in* $\mathbf{N_{MTL2}}$.

Proof By induction on derivations of the cut-free fragment of $\mathbf{HL}^{\forall}_{\mathbf{BCK}}$. All cases are straightforward. For branching rules, we have to apply (EC) because in $\mathbf{HL}^{\forall}_{\mathbf{BCK}}$, the components in a hypersequent are distributed additively while in $\mathbf{N_{MTL2}}$ multiplicatively. And above and below (EC), we have to use ($\multimap$I) and ($\multimap$E) so that the restricted form of (EC) rule in $\mathbf{N_{MTL2}}$ is applicable. ∎

4.5 Discussion and Future Work

This work is technically similar to Danos and Krivine's [38]. One large difference is the existence of (com*i*), (read) and (write) reductions, where terms are passed from processes to other processes. In [38], the only reduction rule involving multiple processes is (dist), which spawns processes that never communicate. Sharing their purpose "interpretation of logical rules as programming instructions" [38], we continue to seek generalization to other substructural logics.

The asynchronous semantics given here is similar to that of a hyper-lambda calculus λ-GD by Hirai [77]. We conjecture that our classical realizability argument here is applicable to λ-GD as well, but for that, due to contraction rule, we have to generalize the statement of adequacy to involve arbitrarily many derivations.

There are recent developments [26, 146] over Curry-Howard correspondence between the linear logic and process calculi. Since their type systems do not incorporate the prelinearity axiom (without modalities ! or ?), we conjecture that we can extend their type systems and their process calculi with the prelinearity axiom representing the same kind of communication schemes as we have shown[4].

Hájek's basic fuzzy logic (BL) is very similar to MTL [28]. Actually from MTL, BL can be obtained by adding $(\varphi \otimes (\varphi \multimap \psi)) \multimap (\psi \otimes (\psi \multimap \varphi))$ (or a symmetric axiom). Thus, we have to find the computational meaning of the additional axiom in order to find a lambda calculus for BL.

4.6 Conclusion

We developed a lambda calculus for MTL2 and analyzed the prelinearity axiom using the classical realizability. The terms typed with prelinearity are asynchronous communication schemes.

[4]Although it is not clear how to implement shared memory in the target process calculi.

Chapter 5

Hypersequents in the Programming Language Haskell

5.1 Summary

We investigate the computational meaning of hypersequents and realize it as a Haskell library. The hypersequent calculus is introduced by mathematical logicians [8, 30, 102] in order to obtain cut-free deduction systems for more logics. In the previous chapters, we proposed using hypersequents for representing asynchronous or synchronous communication. In this chapter, we try implementing hypersequents within a programming language `Haskell` [98]. Logically, a hypersequent means every model satisfies at least one component. Computationally, a hypersequent means every execution makes at least one component successful. We use this analogy for waitfreely communicating threads. This is useful because typical waitfree protocols use the fact that at least one thread can successfully read other thread's information. Throughout this chapter, `Haskell` syntax is assumed[1]. Another purpose of this chapter is confirmation of the proof of Theorem 3.4.3 and Theorem 3.4.7 by implementing it.

5.2 Introduction

5.2.1 Waitfreedom

Waitfreedom is a notion born in the theory of asynchronous communication [72, 93, 125]. Since waitfreedom provides no synchronization among processes, it has served as a basis for comparing different synchronization primitives. The intuition of waitfreedom is simple: a thread cannot wait for another thread. Suppose there are a finite

[1]Under Haskell platform, `cabal install waitfree` will install this library.

number of threads and they can communicate using a store. They can visit the store, put and take things on the store and then leave the store. The problem is that the thread's movements can be arbitrarily slow: they can delay for unspecified amount of time and the threads have no control over the delay. Waitfreedom prohibits a thread from waiting for another; that is, a thread cannot choose to stay at the store until another thread comes to the store; nor can a thread keep visiting the store until another thread comes to the store[2]. The latter restriction can be formalized as existence of a constant natural number k so that no thread visits the store more than k-times in any execution. So, when a thread consumes all permitted visits to the store, the thread has to give up receiving anything yet to come from other threads. We can enforce this restriction by prohibiting looping so that each thread can make steps at most the number of lines in its program.

This notion of waitfreedom is originally imperative: involving reads from and writes to the memory and the ordering of events. In other words, this description is about *how* waitfree computation works but it is hard to deduce from this notion *what* waitfree computation computes. The merit of using a functional programming language, especially a pure one like Haskell, is emphasized if the program describes *what* rather than *how* it computes.

Concrete examples give a useful intuition for modeling what waitfree protocols can compute. Suppose a thread has a value v and another has w. It is waitfreely impossible for the two threads to exchange their possessions. Instead, there is a waitfree protocol for them to ensure that either w is passed to the second thread or v is passed to the first thread. In other words, only one-way communication is guaranteed. We are going to exploit this property when formulating waitfree computation with hypersequents.

5.2.2 Implementing Hyper-Lambda Calculus

In waitfree protocol, two threads can communicate in one-way fashion, and it is decided at execution time which one can send information to the other. This phenomenon can be understood as a set of computations of which at least one is guaranteed to be successful. Such a set of computations can be represented as a hypersequent. `K s a` represents thread `s`'s computation yielding a value of type `a` and `K t b` represents thread `t`'s computation yielding a value of type `b`. The waitfree protocol described at the end of Subsection 5.2.1 witnesses the following derivation because either `t` obtains `b` or `s` obtains `a` after executing the protocol.

[2]Thus a thread cannot busy-wait for the shared memory to be filled.

$$\frac{\texttt{K t a} \quad \texttt{K s b}}{\texttt{K t b} \mid \texttt{K s a}}$$

However, this derivation only contains types but not programs. In this chapter, we are going to propose how to write a program belonging to this type in Haskell.

From this perspective, the merit of this chapter exists in showing the hyper-lambda calculi are implementable. Hypersequents are roughly sequences of typed terms. At first, they were introduced as an apparatus for obtaining cut-elimination theorems for more logics. However, as we have seen in Chapters 2, 3 and 4, we can obtain programming languages out of hypersequent calculi. Thus it is natural to try implementing the programming languages.

5.3 An Example Program

We show an example program written using our library (Figure 5.1). This is a concrete realization of the simplest nontrivial waitfree computation described at the end of Subsection 5.2.1.

The external behavior of the example program. This program can be compiled and executed as:

```
% ghc --make -threaded waitfree_test.hs
[1 of 1] Compiling Main
Linking waitfree_test ...
% ./waitfree_test
```

Then, it spawns two threads listening on port 6000 and 6001. If we connect to the first thread on port 6000 and give input `apple`, the first thread tries to obtain the input given to the other thread but since the first thread cannot wait for the second, fails to do so and aborts. Below, we omit the outputs of `telnet` for clarity.

```
% telnet localhost 6000
0 requiring input: apple
Thread 0 failed to read peer's input.
```

When we connect to the second thread on port 6001 and give another input `orange`, the second thread obtains the input `apple` given to the first thread and displays both inputs.

```
% telnet localhost 6001
```

```
1 requiring input: orange
Thread 1 got: ("orange\r","apple\r")
```

The internal behavior of the example program. Each of threads 0 and 1 writes its input into the shared memory and then tries to read the other thread's input from the shared memory. When a thread tries to read what the other thread has not written, the reader aborts. The reading thread cannot wait for the other thread's write operation.

We assume sequential consistency. Sequential consistency is an illusion sustained by shared memory mechanism that pretends to align all memory operations in a totally ordered sequence. In this case, there are six possible such ordered sequences.

- 0 writes, 0 reads, 1 writes and 1 reads. Then, 1 reads 0's write.
- 0 writes, 1 writes, 0 reads and 1 reads. Both read each other's write.
- 0 writes, 1 writes, 1 reads and 0 reads. Both read each other's write.
- 1 writes, 0 writes, 0 reads and 1 reads. Both read each other's write.
- 1 writes, 0 writes, 1 reads and 0 reads. Both read each other's write.
- 1 writes, 1 reads, 0 writes and 0 reads. Then, 0 reads 1's write.

In any of these six cases, either thread 0 obtains thread 1's input or 1 obtains 0's input.

The source code of the example program. We look at the source code of this sample program in Figure 5.1.

For implementing the above-described external behavior, an obvious thing to do is to establish a TCP connection. This is done by a function called `handle`. The input of `handle` is a `PortID`, which is the port number. The output of `handle` is a `Handle`, which is a TCP connection.

```
handle :: PortID -> IO Handle
handle p = withSocketsDo $ do
  s <- listenOn p
  (h,_,_) <- accept s
  hSetBuffering h NoBuffering
  return h
```

Since this connection establishment should be done by a thread, we prepare a hypersequent containing this computation. For that, we can use our library function `single` that creates a hypersequent containing a single component.

```
prepareHandle :: Thread t =>
                 PortID -> IO (K t Handle :*: HNil)
prepareHandle p = single $ handle p
```

Next the program asks the user to provide an input. In doing that, showing the name of the current thread is useful. When a type `t` is of the type class `Thread`, we can obtain the thread's number via `atid` function (`atid` stands for Abstract Thread ID).

```
readLine :: Thread t =>
            t -> Handle -> IO ((Handle, String), String)
readLine th h = do
  hPutStr h $ (show $ atid th) ++ " requiring input: "
  str <- hGetLine h
  return ((h, str), str)
```

The result of this interaction is stored in a tuple `((h, str), str)`, whose left part contains things to be used locally, and whose right part contains things to be used remotely in the other thread. The handle `h` is kept locally at the thread while the obtained input `str` is duplicated.

This distinction of local and remote usage is apparent in the type of `comm` function, which provides the simplest waitfree communication.

```
comm :: (Thread s, Thread t, HAppend l l' l'') =>
         IO (K t (b,a) :*: l)
         -> IO (K s (d,c) :*: l')
         -> IO (K t (b,c) :*: K s (d,a) :*: l'')
```

The `comm` function assumes that the types `s` and `t` are `Thread`s and that a hypersequent `l''` is the concatenation of `l` and `l'`. Under these assumptions, the function takes two hypersequents in `IO` monad and returns a hypersequent in `IO` monad. The first argument says either `t`'s computation of a value of type `(b,a)` or one of the components of the hypersequent `l` succeeds. The second argument says the same thing for `s`'s `(d,c)` and `l'`. If any component in `l` or `l'` is successful, that component witnesses the resulting hypersequent. Otherwise, when thread `t` computes `(b,a)` value and `s` computes `(d,c)` value successfully, the `comm` function guarantees either `t` obtains the `c` value or `s` obtains the `a` value. The `b` value and the `d` value are only used locally. Without this distinction of local and remote usage, the TCP connections would be communicated together with the inputs so that the connections are not closed until all threads terminates.

When we draw the type of `comm` in the hypersequent style,

$$\frac{\texttt{IO (K t (b,a) :*: l)} \qquad \texttt{IO (K s (d,c) :*: l')}}{\texttt{IO (K t (b,c) :*: K s (d,a) :*: l'')}}$$

we see some similarities and differences from the hyper-lambda calculus in Chapter 3. When we ignore `b` and `d`, then the rule is the same as ij-com rule in Figure 3.2 except that the type of `comm` does not have contexts.

The final agenda is to output the obtained inputs. This is done by the `printTaken` function. Similarly to `readLine` function, the `printTaken` function takes the thread and the result of the last computation.

Everything is wrapped up into a hypersequent `content` of the type

```
IO (K ZeroT () :*: K (SucT ZeroT) () :*: HNil)
```

and it is `execute`d. Here, `ZeroT` means thread 0 and `SucT ZeroT` means thread 1. The content of `execute` function is explained in the next section.

5.4 Implementation of the Library

5.4.1 What is a Hypersequent

We implemented hypersequents as heterogeneous collections by Kiselyov et al. [88] with a slight modification. We required each element of such a heterogeneous collection to be a thread's computation. We expressed threads as type level natural numbers, which is obtained by simplifying heterogeneous lists. Each thread is expressed as a singleton type containing a single value. When that value is passed to the `atid` function, the function returns an `AbstractThreadId`, which is tentatively defined to be `Int`.

A type `t` can be of type class `Thread` only if there is a constant called `t` of type `t`.

```
class Thread t where
    t :: t
    atid :: t -> AbstractThreadId
```

We declare a type called `ZeroT`, which have a single value called `ZeroT` (it is a custom of the Haskell community to use the same name for a type and its unique constructor). The type `ZeroT` is of type class `Thread` after defining constant `t` to be the single value `ZeroT`. The type `ZeroT` is going to be used as the label for thread 0.

```
data ZeroT = ZeroT
instance Thread ZeroT where
    t = ZeroT
    atid ZeroT = 0
```

We are going to allow users to use as many threads as they want. After `ZeroT`, we inductively define infinitely many types of type class `Thread`. For type `t`, we define a type `SucT t`. A value of `SucT t` is of the form `SucT t` where `t` is a value of type `t`. Especially, when type `t` is of type class `Thread`, the new type `SucT t` is also of type class `Thread`. The type class `Thread` requires a constant `t` of `SucT t`, which we define to be `SucT t` where value `t` of type `t` is the constant provided by our assumption that type `t` is of type class `Thread`.

```
data SucT t = SucT t
instance Thread t => Thread (SucT t) where
    t = SucT t
    atid (SucT x) = succ $ atid x

type AbstractThreadId = Int
```

Using these, we have many types of type class `Thread`: `ZeroT`, `SucT ZeroT` and so on. This technique is called type level natural numbers.

Thread `t`'s computation of type `a` is represented as `K t a`. Internally, it is a pair of thread and a computation of `JobStatus a`. A `JobStatus` can be `Having a`, `Done TryAnotherJob` or `Done Finished`. `Having a` shows the thread's computation is not yet finished and as the intermediate result the thread have a value of type `a`. `Done TryAnotherJob` shows that the thread has stopped after failing to read shared memory content. `Done Finished` shows that the thread has stopped.

```
newtype K t a = K (t, IO (JobStatus a))

data JobStatus a = Having a | Done ThreadStatus
data ThreadStatus = TryAnotherJob | Finished
```

A hypersequent is either `HNil` or `HCons (K t e) l` where `l` is a hypersequent. This inductive definition realizes the idea of a heterogeneous list of threads' computations. This construction is an adaptation of [88]. First we declare a type class called `HyperSequent`.

```
class HyperSequent l
```

Second, we define the empty hypersequent called `HNil` of type `HNil` of type class `HyperSequent`.

```
data HNil = HNil
instance HyperSequent HNil
```

Third, we define the cons operation of hypersequent. `HCons e l` is of type `HCons e l` when `e` is of type `e` and `l` is of type `l`. Especially, when type `l` is of type class `HyperSequent`, the type `HCons (K\,t\,e) l` is also of type class `HyperSequent` whatever the type `e` is.

```
data HCons e l = HCons e l
instance HyperSequent l =>
  HyperSequent (HCons (K t e) l)
```

Finally, we define an abbreviation for type `HCons e l`.

```
infixr 5 :*:
type e :*: l = HCons e l
```

In our framework, every hypersequent is going to be contained in `IO` monad so that hypersequent derivations can contain MVar cells. The MVar cells are used for implementing the communication rule. Although we use MVar cells, which are capable of supporting full synchronization among threads, we only use a limited set of functions that only allows waitfree communication. For instance, we do not use `takeMVar` or `putMVar` because they are blocking operations.

5.4.2 How to Execute a Hypersequent

Of course we can execute a hypersequent after we construct it. The execution takes three steps: preparing computations, spawning threads and waiting for them.

After a hypersequent derivation is constructed, it is turned into a list of local computations. A value of type `L` is internally used to represent a piece of local computation. Each piece of computation results in either `TryAnotherJob` or `Finished`. These decide whether the thread continues to operate or stops.

```
type L = (AbstractThreadId, IO ThreadStatus)
```

Local computations are then transformed into a finite map from thread identifiers to channels containing computations. Each thread reads the list for its own identifier and executes the computations in the list one by one. A type `l` can be of type class `Lconvertible` only if a value of type `l` can be converted into a list of `L` values.

```
class Lconvertible l where
    htol :: l -> [L]
```

Any value of type `HNil` is converted into an empty list of `L` values.

```
instance Lconvertible HNil where
    htol _ = []
```

A longer hypersequent is also convertible into a list of `L` values.

```
instance (Thread t, Lconvertible l) =>
 Lconvertible (HCons (K t ThreadStatus) l) where
    htol (HCons (K (th, result)) rest) =
      (atid th, fmap jth2th result) : htol rest

jth2th :: JobStatus ThreadStatus -> ThreadStatus
jth2th (Having x) = x
jth2th (Done x) = x
```

Since a list of type `[L]` contains pieces of computation for possibly multiple threads, the next thing is demultiplexing the list into many lists each of which serves jobs to one thread. We construct a list of computations for each thread. The lists have type `JobChannel`. The `JobChannel`s are put in a finite map called `JobPool`. The `JobPool` maps each `AbstractThreadId` to a `JobChannel`. Later, the worker threads look at this map to take jobs.

```
type JobChannel = [IO ThreadStatus]

type JobPool =
    Map.Map AbstractThreadId JobChannel
```

The actual conversion is done by `constructJobPool`, which acts inductively on a list of type `[L]`.

```
constructJobPool :: [L] -> JobPool
constructJobPool [] = Map.empty
constructJobPool ((aid, action) : tl) =
  Map.insertWith (++) aid [action] rest
     where rest = constructJobPool tl
```

At last, threads are spawned. Each thread's operation is defined in `worker` function. It takes two arguments. The first argument is a channel containing jobs. The other argument is an MVar cell for telling termination to the main thread. Although the worker threads do not wait for any other worker thread, the main thread waits for the worker threads. The worker tries consuming jobs in the provided `JobChannel` one by one until one succeeds or all fails. After the worker consumes all jobs in the provided `JobChannel`, the worker puts `()` into the provided MVar cell so that the main thread can notice that the worker is idling.

```
worker :: JobChannel -> MVar () -> IO ()
worker [] fin = tryPutMVar fin () >>= \_ -> return ()
worker (hd : tl) fin = do
  result <- hd
  case result of
    TryAnotherJob -> worker tl fin
    Finished -> tryPutMVar fin () >>= \_ -> return ()
```

When the main thread spawns the worker threads, it has their `ThreadId`s and the termination MVar cells in a finite map. After spawning the worker threads, the main thread waits for each worker thread to fill the MVar and then kills the worker thread. A value of type `ThreadPool` maps a value of `AbstractThreadId` into a value of `ThreadId` and an MVar. A value of type `ThreadId` is provided by the Haskell `Concurrent` library and a value of `ThreadId` can be used to kill a thread. The `MVar` is filled when the worker thread finishes the computation and then the main thread can see the worker thread has finished.

```
  type ThreadPool = Map.Map AbstractThreadId (ThreadId, MVar ())
```

The main thread uses `threadSpawn` to register a new thread in the thread pool using an `AbstractThreadId` and a `JobChannel`.

```
threadSpawn :: AbstractThreadId -> JobChannel -> IO ThreadPool -> IO ThreadPool
threadSpawn aid ch p = do
    p' <- p
    fin <- newEmptyMVar
    thid <- forkIO $ worker ch fin
    return $ Map.insert aid (thid, fin) p'
```

Then, the main thread waits for all worker threads and kills them.

```
waitThread :: ThreadPool -> IO ()
waitThread = Map.fold threadWait $ return ()
```

```
threadWait :: (ThreadId, MVar ()) -> IO () -> IO ()
threadWait (thid, fin) w = do
    readMVar fin
    killThread thid
    w
```

At last, everything is wrapped up into a function called `execute`.

```
execute :: Lconvertible l => IO l -> IO ()
execute ls = do
  ls >>= run . constructJobPool . htol >>= waitThread

run :: JobPool -> IO ThreadPool
run = Map.foldrWithKey threadSpawn $ return Map.empty
```

Since all hypersequents are of types belonging to type class `Lconvertible`, they can be `execute`d.

5.4.3 How to Construct a Hypersequent

Since a hypersequent represents a finite multiset of computations of which at least one succeeds, there cannot be an empty hypersequent. Thus, the simplest hypersequent is a hypersequent consisting of a single component. This can be built with `single` function. The ingredient computation is not of type `IO a` but of type `t -> IO a` so that the computation can display the thread `t`'s `AbstractThreadId`.

```
single :: Thread t => (t -> IO a) -> IO (K t a :*: HNil)
single f = return $ HCons (remote $ f t) HNil
  where remote y = K (t, fmap Having y)
```

As we want to regard a hypersequent as a multiset of components, we allow permutation and concatenation of hypersequents. First, `exchange` function changes the positions of the first two components:

```
exchange :: K t a :*: K s b :*: l -> IO (K s b :*: K t a :*: l)
exchange (HCons x (HCons y rest)) = return $ HCons y $ HCons x rest
```

Second, `cycling` functions puts the last component in the first position. In order to implement this function, we defined a type class called `HLast`. The type class instance `HLast l a heads` means that the hypersequent `l` is a concatenation of `heads` and a singleton list made of `a`. In other words, the last element of `l` is `a` and the rest is `heads`. This tactic of defining a type class is taken from the paper of heterogeneous collections [88]. By `exchange` and `cycling`, we can perform all permutations of hypersequents[3].

First, we declare the type class `HLast`. Here we use functional dependencies [84] to specify that the choice of type `l` uniquely determines the types `a` and `heads` that satisfies `HLast l a heads` (if there exists such `a` and `heads` at all). In other words, `HLast` is actually a partial function that takes type `l` and returns types `a` and `heads`.

```
class HLast l a heads | l -> a, l -> heads
 where hLast :: l -> (a, heads)
```

The `HLast` is defined on hypersequents inductively.

```
instance HLast (HCons a HNil) a HNil
    where hLast (HCons x HNil) = (x, HNil)

instance (HLast (HCons lh ll) a heads) =>
  (HLast (HCons b (HCons lh ll)) a (HCons b heads))
    where hLast (HCons y rest) =
              case hLast rest of
                (x, oldheads) -> (x, HCons y oldheads)
```

Using this type class `HLast`, we can define `cycling` function, which moves the last element of a hypersequent to the first position.

```
cycling_ :: HLast l a heads => l -> HCons a heads
cycling_ lst = case hLast lst of
                (last_, heads) -> HCons last_ heads

cycling :: HLast l last heads => IO l -> IO (HCons last heads)
cycling = fmap cycling_
```

Finally, we provide concatenation of hypersequents with `follows` function. Again, we prepare a type class called `HAppend` for this. This technique is also from Kiselyov et al. [88].

```
follows :: HAppend l l' l'' => IO l -> IO l' -> IO l''
follows l0 l1 = do
  h0 <- l0
```

[3]This implementation of permutations comes from Tatsuya Abe's implementation of a typed lambda calculus for modal logic K. Tatsuya Abe showed the Agda [24] implementation to the author in Tokyo.

```
  h1 <- l1
  return $ hAppend h0 h1

class HAppend l l' l'' | l l' -> l''
 where hAppend :: l -> l' -> l''

instance HyperSequent l => HAppend HNil l l
 where hAppend HNil = id

instance (HyperSequent l, HAppend l l' l'')
    => HAppend (HCons x l) l' (HCons x l'')
 where hAppend (HCons x l) = HCons x. hAppend l
```

In addition to simple permutations of hypersequents, we have another structural rule called external contraction. When a hypersequent begins with two components of the same type, they can be merged into one. The resulting component represents the computation of trying the two original components one by one until one of them is successful or all components fail. The type of `choice` reveals that the two components of the same type `K t a` are squashed into a single component.

```
choice :: Thread t => K t a :*: K t a :*: l -> IO (K t a :*: l)
choice (HCons (K (_, comp0))
  (HCons (K (_, comp1)) rest)) =
  return $ HCons (K (t, result)) rest
  where
    result = do
      r0 <- comp0
      case r0 of
        Having a -> return $ Having a
        Done TryAnotherJob -> comp1
        Done Finished -> return $ Done Finished
```

It is possible to compose a piece of local computation with another piece of computation and obtain a new piece of local computation. For that purpose, we prepare `extend` function.

```
extend :: Thread t => (K t a -> IO (JobStatus b)) -> K t a -> K t b
extend trans r = K (t, trans r)
```

Such transformations of local computation can be changed into transformations of hypersequents.

```
infixr 4 -*-

(-*-) :: (Thread t, HyperSequent l, HyperSequent l') =>
            (t -> a -> IO b) -> (l -> IO l') ->
            HCons (K t a) l -> IO (HCons (K t b) l')
(-*-) = progress_ . extend . peek . lmaybe
  where
    lmaybe _ _  (Done x) = return (Done x)
    lmaybe f th (Having x) =  do
      y <- f th x
      return $ Having y

progress_ :: (HyperSequent l, HyperSequent l') =>
            (a -> b) -> (l -> IO l') -> HCons a l ->
            IO (HCons b l')
progress_ hdf tlf (HCons ax bl) = do
  newtl <- tlf bl
  return $ HCons (hdf ax) newtl
```

In such transformations, the extending function must receive the result of the previous function. This is achieved with the help of `peek` function.

```
peek :: Thread t => (t -> JobStatus a -> IO b) -> K t a -> IO b
peek f (K (th, content)) = content >>= f th
```

The most interesting library function `comm` makes two threads compute something, write it to the shared memory, reads peer's write if possible and then continue computation. We split a thread's computation into two parts: one part up to the write and the other from the read operation. The former part is hidden in the local computation whereas the latter part is stored in the hypersequent so that the types of the latest computation are visible to the user of the library. `comm` stands for communication. `comm` combines two hypersequents each containing a communicating component. It can be used in the form `comm hypersequent1 error1 hypersequent2 error2` where `error1` and `error2` specifies what to do in case of read failure.

```
comm :: (Thread s, Thread t, HAppend l l' l'') =>
        IO (HCons (K t (b,a)) l)
         -> (t -> b -> IO ThreadStatus)
         -> IO (HCons (K s (d,c)) l')
         -> (s -> d -> IO ThreadStatus)
        -> IO (K t (b, c) :*: (K s (d, a) :*: l''))
comm x terror y serror = do
  HCons (K (taT, tba)) l <- x
  HCons (K (scT, sdc)) l' <- y
  abox <- newEmptyMVar
  cbox <- newEmptyMVar
  return $ let
      tbc = comm_part tba abox cbox terror taT
      sda = comm_part sdc cbox abox serror scT
    in
    HCons (K (taT, tbc))
      (HCons (K (scT, sda)) (hAppend l l'))
    where
      comm_part tba wbox rbox err th = do
            maybeba <- tba
            case maybeba of
              Done thStatus -> return $ Done thStatus
              Having (tb, ta) -> do
                _ <- tryPutMVar wbox ta    -- writing
                cval <- tryTakeMVar rbox   -- reading
                case cval of
                  Nothing -> do
                    terror_result <- err th tb
                    return $ Done terror_result
                  Just cva -> return $ Having (tb, cva)
```

In the sixth and seventh lines counted from the bottom, both threads perform write and then read so that at least one direction of communication is successful.

5.5 Capturing Waitfreedom

Since this chapter presents a library for waitfree computation, it is desirable that only waitfree computation can be implemented using this library (soundness), and all waitfree computation can be implemented using this library (completeness).

5.5.1 Soundness

In the library code, the worker threads only use non-blocking `MVar` operations (a blocking operation `readMVar` is used but by the main thread). This ensures that only waitfree computation can be implemented using this library unless the user explicitly uses other communication primitives from outside of this library.

5.5.2 Completeness

We show the completeness by solving a problem called the participating set problem appearing in Borowsky and Gafni [23]. In the participating set problem, threads obtain no input except their own id's. Each thread i outputs a set S_i of thread id's satisfying the following conditions:

1. $i \in S_i$,
2. $S_i \subseteq S_j$ or $S_j \subseteq S_i$ for any i, j
3. $i \in S_j$ implies $S_i \subseteq S_j$.

We can solve any waitfree protocol by repeating solving a finite number of the participating set problem instances.

Intuitively, this can be achieved by making every thread write its id to the shared memory and then read others' writes from the shared memory. The results are determined by the speed competition among the threads. Thread i's output S_i contains those threads that has already written by the time i reads. However, we have to decompose this intuitive solution into pieces of two-thread communication in order to implement the solution using our library.

As an example, we treat the three-thread case. The three threads are named `ZeroT`, `FirstT` and `SecondT`.

First, each thread computes its own id and stores it in a tuple.

```
hOwnId :: Thread t => IO (K t (t, t) :*: HNil)
```

Then, `ZeroT` and `FirstT` compete. This results in the following hypersequent, which means that either `ZeroT` obtains `FirstT`'s id or `FirstT` obtains `ZeroT`'s id.

```
IO (K ZeroT (ZeroT, FirstT) :*: K FirstT (FirstT, ZeroT) :*: HNil)
```

Before `SecondT` is put in consideration, we make the first two threads to duplicate their possessions so that they can retain the possessions and communicate the possessions at the same time.

```
twoBeforeComm :: IO ( K ZeroT ((ZeroT, FirstT), (ZeroT, FirstT)) :*:
                      K FirstT ((FirstT, ZeroT), (FirstT, ZeroT)) :*:
                      HNil)
```

After `SecondT` competes with `ZeroT`, we obtain the following hypersequent. `FirstT`'s possessions are not changed, but `ZeroT` and `SecondT` tries to exchange their possessions.

```
three__ :: IO
  (K FirstT ((FirstT, ZeroT), (FirstT, ZeroT)) :*:
   K ZeroT ((ZeroT, FirstT), SecondT) :*:
   K SecondT (SecondT, (ZeroT, FirstT)) :*: HNil)
```

Finally, `SecondT` competes with `FirstT`. This makes a hypersequent with any component containing all thread's ids. Since at least one component succeeds, at least one thread obtains all threads' ids.

However, the participating set problem requires more. The following output does not conform to the problem although one thread has obtained all threads' ids. Either $S_0 \subseteq S_2$ or $S_2 \subseteq S_0$ is required but neither is satisfied.

```
Thread 1: obtained [1,0,2]
Thread 0: obtained [0]
Thread 2: obtained [2]
```

In this case threads 0 and 2 have to compete in order to decide which is slower. In general, either `ZeroT`–`SecondT` or `FirstT`–`SecondT` competition is required so we do both and concatenate the three competitions as one derivation. Both of the two-thread competitions are similar to the simplest nontrivial waitfree protocol already explained so we omit the details. The whole program is shown in Figure 5.2.

```
main :: IO ()
main = execute $three ‘follows‘ twoMid ‘follows‘ twoLast
```

5.6 Comparison with Gödel-Dummett Logic

5.6.1 Similarity

There are many functional programming languages (OCaml, Haskell, Clean etc.) that employ intuitionistic logic for their type systems. The hypersequent formulation that we employ is very similar to that of Gödel-Dummett logic, which is among intermediate logics [137] between intuitionistic and classical logics.

Gödel-Dummett logic [43] was originally introduced as a logic obtained by adding axioms of the form $(\varphi \to \psi) \vee (\psi \to \varphi)$ to intuitionistic propositional logic.

Avron [8] proposed an alternative formulation of the same logic. Instead of the axioms $(\varphi \to \psi) \vee (\psi \to \varphi)$, he included the "communication rule" on hypersequent derivations:

$$\frac{\mathcal{H}_0 \mid \Gamma_0, \Delta_0 \vdash \varphi_0 \qquad \mathcal{H}_1 \mid \Gamma_1, \Delta_1 \vdash \varphi_1}{\mathcal{H}_0 \mid \mathcal{H}_1 \mid \Gamma_0, \Delta_1 \vdash \varphi_0 \mid \Gamma_1, \Delta_0 \vdash \varphi_1}$$

where Γ_i and Δ_i are finite sets of formulae. This communication rule inspired the author to model waitfree computation using hypersequents by using a rule

$$\frac{\mathcal{H}_0 \mid \texttt{K t a} \qquad \mathcal{H}_1 \mid \texttt{K s b}}{\mathcal{H}_0 \mid \mathcal{H}_1 \mid \texttt{K t b} \mid \texttt{K s a}}$$

Avron [8] has proved that a sequent derivable with cut rules is also provable without cuts (Gentzen's proof adopted). However, he still wanted to see what the corresponding typed lambda calculus looks like, saying "it seems to us extremely important to determine the exact computational content of them [intermediate logics]—and to develop corresponding 'λ-calculi' " [8]. Since we are working on a constructive logic, it is natural to ask what chooses the left or right for a term of the type $(\varphi \supset \psi) \vee (\psi \supset \varphi)$. The answer is actually the execution time scheduling among threads.

5.6.2 Difference

Gödel-Dummett logic does not have modalities. Its formulae are the same as intuitionistic or classical logic. The only difference of Gödel-Dummett logic from intuitionistic logic is that more formulae are provable in Gödel-Dummett logic than in intuitionistic logic. So, if we want to encode Gödel-Dummett logic faithfully, we do not have to introduce modalities such as `K t`. Even without such modalities, we can regard each component of a hypersequent as a thread. However, If we abandon modalities and regard components of a hypersequent as threads, the participating set problem cannot be solved because the communication rule splits the threads into two separate groups and any prior inter-group communication is prohibited.

This restriction would be OK if we were only interested in making sure that at least one thread knows every other thread. However, since the participating set problem does not allow any two threads to be ignorant of each other, the `comm` rule cannot provide enough communication power to solve the participating set problem for the components of a hypersequent if we do not have the modalities.

5.7 Related Work

5.7.1 Computational Content of Hypersequents

Fermüller [48] developed a so-called parallel dialogue games. It is based on Lorenzen dialogue . It is basically proof searching from bottom to up with player supposed to know the answer proof tree and opponent directing for different parts of the proof. Since the game rules are translation of hypersequent calculus, the global game state has local parts. And the player can, for example, duplicate a local part into two. This is essentially different from the typed lambda calculi computation because the games give meanings to only normal form proofs while the typed lambda calculi give meanings to all proofs.

5.7.2 High Level Treatment of Concurrency

Erlang [5] is a programming language designed for concurrency using the Actor model by Hewitt et al. [73]. Since the Actor model separates the sender of a message from the message itself, it provides some asynchrony. However, in the Actor model and in Erlang, a thread can wait for a message. Thus, these do not provide waitfree communication.

Asynchronous π-calculus by Honda and Tokoro [79] is a fragment of π-calculus. It is similar to the Actor Model in that it is impossible to do something after sending a message. However, asynchronous π-calculus allows a process to wait for a channel to deliver a message. Thus, asynchronous π-calculus is not waitfree.

Concurrent ML [123] "provides a high-level model of concurrency." However, its basic communication primitives `accept` and `send` are blocking operations so that there is no fragment of the language capturing waitfreedom.

Brown [25] proposed a combinator library for message-passing programming in Haskell. However, this work is also for synchronous communication with blocking operations and there is no way of obtaining a waitfree fragment of this library.

Join patterns (first proposed as the join calculus by Fournet and Gonthier [51]) allow consuming messages from a group of channels simultaneously. This involves waiting for a group of channels so that join patterns do not capture waitfreedom either.

In multi-core and multi-processor environment, synchronous communication is time-consuming. Compared with synchronous communication, waitfreedom can be implemented in a less time-consuming manner. Since our library interface captures waitfreedom, it is possible to build a less time-consuming implementation for this interface although the current implementation relies on MVar cells, which internally use spin locks.

5.8 Future Work and Conclusions

Ciabattoni, Galatos, and Terui [30] showed a class of logics can be defined using hypersequent derivation rules. It will be interesting to see what kind of computation is represented by these logics.

The library presented in this chapter deals with both waitfree communication and thread management. It would be better if we dealt with these different tasks separately. We seek to expand the notion of threads into more complicated processes so that we can treat dynamic forking and merging threads rather than a finite set of threads specified at compile-time.

Although we were able to implement the participating set problem, the implementation is not short or symmetric. There seems to be room for more elegant abstraction for symmetric protocols such as a rule containing more than two threads and a more direct embedding of the participating set problem. The main challenge is encoding the n-party participating set problem in the Haskell type system uniformly with respect to n.

Moreover, in order to obtain performance, we have to change the runtime system of Haskell. Since the current implementation uses MVar mechanism that involve spin-locks, it must be unnecessarily slow. In theory, waitfree computation can be implemented without locks. In practice also, with sequential consistency, we can implement waitfree computation without locks.

One possible extension of our library is addition of an "all-possible-results" mode. When we execute a program under this mode, the library explores all possible executions and returns what can happen.

Notwithstanding, since we have shown that hypersequent based waitfree computation is implementable. This at least supports the statement of Theorem 3.4.3 in Chapter 3. Also as a side-effect, we obtained a type-checking implementation of a hyper-lambda calculus.

```
handle :: PortID -> IO Handle
handle p = withSocketsDo $ do
  s <- listenOn p
  (h,\_,\_) <- accept s
  hSetBuffering h NoBuffering
  return h
prepareHandle :: Thread t => PortID -> IO (K t Handle :*: HNil)
prepareHandle p = single $ handle p

readLine :: Thread t => t -> Handle -> IO ((Handle, String), String)
readLine th h = do
  hPutStr h $ (show $ atid th) ++ " requiring input: "
  str <- hGetLine h
  return ((h, str), str)
readH :: Thread t => PortID -> IO (K t ((Handle, String), String) :*: HNil)
readH p = prepareHandle p >>= (readLine -*- return)

printTaken :: Thread t => t -> ((Handle, String), String) -> IO ()
printTaken th ((h, selfs), peers) = do
        hPutStrLn h $ (show $ atid th) ++ " got: " ++ show (selfs, peers)
        return ()

twoPrints :: K ZeroT ((Handle, String), String) :*:
             K (SucT ZeroT) ((Handle, String), String) :*: HNil
              -> IO (K ZeroT () :*: K (SucT ZeroT) () :*: HNil)
twoPrints = printTaken -*- printTaken -*- return

rerror :: Thread t => t -> (Handle, a) -> IO ()
rerror th (h, _) = hPutStrLn h $ "Thread " ++ (show $ atid th) ++
                   " failed to read peer's input."

content ::  IO (K ZeroT () :*: K (SucT ZeroT) () :*: HNil)
content = comm (readH $ PortNumber 6000) rerror (readH $ PortNumber 6001) rerror
          >>= twoPrints
main :: IO ()
main = execute content
```

Figure 5.1: An example program. `:*:` delimits components. Library imports are omitted. This program spawns two threads each waiting for a TCP connection. The two threads do waitfree communication, so the slowest thread obtains the inputs for all threads.

```
failOut :: t -> a -> IO ThreadStatus
failOut _ _ = return TryAnotherJob
hOwnId = single $ return (t, t)
putWithName :: Thread t => t -> String -> IO ()
putWithName th content = putStrLn $ "Thread " ++ (show $ atid th) ++ ": " ++ content
putResult :: Thread t => t -> String -> IO ThreadStatus
putResult th str = do
  putWithName th $ "obtained " ++ str
  return Finished
putOneResult :: (Thread t, Thread s) => s -> t -> IO ThreadStatus
putOneResult owner content = putResult owner $ show $ [atid content]
putTwoResults :: (Thread s, Thread t, Thread u) => u -> (s,t) -> IO ThreadStatus
putTwoResults owner (c0, c1) = putResult owner $ show $ [atid c0, atid c1]
two :: (Thread s, Thread t) => IO (K s (s, t) :*: (K t (t, s) :*: HNil))
two = comm hOwnId failOut hOwnId failOut
twoBeforeComm :: IO (K ZeroT ((ZeroT, FirstT), (ZeroT, FirstT)) :*:
                     K FirstT ((FirstT, ZeroT), (FirstT, ZeroT)) :*: HNil)
twoBeforeComm = two >>= (duplicateTwo -*- duplicateTwo -*- return)
  where duplicateTwo _ x = return (x,x)
printThreeResults0 :: (Thread u, Thread s, Thread t, Thread v) =>
                      u -> (s,(t,v)) -> IO ThreadStatus
printThreeResults0 owner (c0, (c1, c2)) = putResult owner $ show $
  [atid c0, atid c1, atid c2]
printThreeResults1 owner ((c0, c1), c2) = putResult owner $ show $
  [atid c0, atid c1, atid c2]
three :: IO (K FirstT ThreadStatus :*: K SecondT ThreadStatus :*:
             K ZeroT  ThreadStatus :*: K SecondT ThreadStatus :*: HNil)
three = comm (cycling $ comm twoBeforeComm putTwoResults hOwnId failOut)
        putTwoResults hOwnId failOut
        >>= (printThreeResults1 -*- printThreeResults0 -*- printThreeResults1
        -*- printThreeResults0 -*- return)
twoLast :: IO ((K FirstT ThreadStatus) :*: (K SecondT ThreadStatus) :*: HNil)
twoLast = comm hOwnId putOneResult hOwnId putOneResult >>=
          (putTwoResults -*- putTwoResults -*- return)
twoMid :: IO ((K ZeroT ThreadStatus) :*: (K SecondT ThreadStatus) :*: HNil)
twoMid = comm hOwnId putOneResult hOwnId failOut >>=
         (putTwoResults -*- putTwoResults -*- return)
main = execute (three `follows` twoMid `follows` twoLast)
```

Figure 5.2: An implementation for participating set problem [23] for three threads.

Chapter 6

Conclusions

6.1 Overview

We gave mainly two inventions: hyper-lambda calculi and a new axiomatization of Abelian logic. Since we are just beginning to understand the computational semantics of some concrete examples of substructural logics, the remaining space is vast. Reflecting the wide applications of substructural and especially intermediate logics, our conclusions and variety of future work span from mathematical logic, computer science and philosophy.

6.2 From Logical Perspectives

6.2.1 Generalization

We have seen two particular hyper-lambda calculi for two logics. Ciabattoni et al. [30] classified axioms according to their syntactic complexities and identified classes of axioms that can be translated into structural sequent calculus rules and structural hypersequent calculus rules. According to their classification, the prelinearity axiom $(\varphi \multimap \psi) \oplus (\psi \multimap \varphi)$ and the Amida axiom $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$ belong to $\mathcal{P}_2$, a class whose elements can be translated into a finite set of hypersequent structural rules. Other classes $\mathcal{N}_2$ and $\mathcal{P}_3$ of axioms can also be translated into finite sets of hypersequent structural rules, thus, we expect the technique of hyper-lambda calculi applicable to logics with these axioms on top of **FLe**.

One particular logic worth trying is the logic characterized by Kripke models with bounded width [29]. Since Gödel-Dummett logic is a special case of the bounding width 1, the generalization of width k will provide waitfree computation on weaker shared memory consistency. Further, an ambitious goal is to develop a general framework with which we can develop hyper-lambda calculi for all logics characterized by axioms in class $\mathcal{P}_3$. Another target is the logic with the weak excluded-middle $\neg\varphi\vee\neg\neg\varphi$ on top of intuitionistic logic. Since the cut-elimination proof in [30] is algebraic, we are yet to know the computational meaning of the cut-elimination. Aforementioned framework would clarify the computational meaning of the cut-elimination of hypersequent calculi.

6.2.2 The Amida Calculus

As we saw in Subsection 2.8.2, the deduction system underlying the Amida calculus does not enjoy cut-elimination in the strict sense. Moreover, since the Amida calculus characterizes Abelian logic (Theorem 2.5.1), there are some inhabited types that are hard to justify constructively; for example a linear version of the excluded middle $\varphi \oplus (\varphi \multimap \mathbf{1})$. The first step would be removing additives and studying the multiplicative fragment of the Amida calculus.

6.2.3 Gödel-Dummett Logic

In logical point of view, the lambda calculus in Chapter 3 is too complicated for Gödel-Dummett logic. For example, the notion of processes are actually not needed to study Gödel-Dummett logic. However, with the help of these additional constructions, we were able to identify the computational nature of Gödel-Dummett logic as that of waitfree computation. In order to exploit our discovery, we have to extend the characterization to the realm of proof searching. That is, designing a logic programming language using waitfree communication during parallel proof searching.

6.2.4 More General Fuzzy Logics

Since we have identified the computational content of the prelinearity axiom and obtained a lambda calculus for monoidal t-norm logic, we are in a good position to study fuzzy logics in general. After all, monoidal t-norm logic is the weakest fuzzy logic. Thus, there is a possibility to obtain a lambda calculus for any fuzzy logic by extending the lambda calculus in Chapter 4. The next target is Łukasiewicz logic. Łukasiewicz logic is so similar to ordinary logics that there have been attempts to formalize mathematics on it [67, 68, 148]. Metcalfe et al. [103, Theorem 9] showed that a fragment of Abelian logic coincides with Łukasiewicz logic.

6.3 From Computer Science Perspectives

6.3.1 Implementation

In Chapter 5, we implemented a hyper-lambda calculus for waitfree computation on top of Haskell [98]. The implementation is a prototype for showing that the design of λ-GD is implementable. Internally, the implementation still uses locks but we should remove locks if we try to put the library to performance sensitive uses. In order to remove locks, we probably have to modify the runtime system of Haskell. Externally, the implementation lacks primitive support for more than two party communication. Although we can implement arbitrary n-party waitfree communication using the current two-party primitive, having a primitive serving more parties is desirable. In order to provide more than two-party primitives, we have to define a more complicated type class, which we expect to be achievable.

6.3.2 Further Implementation

Since Abelian logic is incompatible with contraction or weakening, straightforward implementation the Amida calculus on top of Haskell or OCaml would not be a good way to exploit the safety of the Amida calculus. Although Clean [113] offers uniqueness types, uniqueness types only reject contraction but accept weakening, so Clean is not suitable either.

One promising framework on which to implement the Amida calculus is linear ML[1], whose type system is based on linear logic. Another way is using the type level programming technique of Haskell. Imai et al. [83] implemented session types on top of Haskell using the type level programming technique using `Session` monad. Since Haskell types can contain arbitrary trees of symbols, they were able to encode session type information in Haskell types. Logically, this corresponds to having atomic formulae with complicated structure so that we can encode session information in atomic formulae. The advantage would be usability of existing Haskell infrastructure including the optimizing compilers, execution environments, and libraries. The disadvantage, which the first author of [83] told me personally, is the compiler's cryptic error messages. Since the used Haskell compiler is not aware of session types and just reports pattern-matching errors in the encoding of session types, the error messages are about the encoding but not about the originally intended session types.

[1]Although there are no publications available, there is an implementation at `https://github.com/pikatchu/LinearML` .

6.3.3 Reasoning about Hyper-Lambda Terms

The Amida edges introduced in Section 2.6 can be applied to other methods than proof nets. For example, in the game semantics of lambda calculi [86], strategies are represented by Nakajima trees [112]. Then we can expect the Amida edges on Nakajima trees to represent a strategy of the game semantics for Abelian logic. Also, the Amida edges works on paths, it should be compatible to the path-based semantics by Danos and Regnier [39]. Since the Amida edges define permutation on the ends of edges in a proof net, and hence can be represented as a matrix, it should be compatible to Geometry of Interaction [60].

6.3.4 Mixing Synchronous and Asynchronous Communication

Our hyper-lambda calculus in Chapter 2 incorporates synchronous communication and another hyper-lambda calculus in Chapter 3 incorporates asynchronous communication. In order to unify these two paradigm, the most straightforward way is to nest hypersequents

$$t_0 \mathbin{\rule[-2pt]{2pt}{12pt}} (t_1 \mathbin{\rule[-2pt]{0.5pt}{6pt}} t_2) \mathbin{\rule[-2pt]{2pt}{12pt}} t_3 \ .$$

where t_1 and t_2 are combined conjunctively but other delimiters are interpreted disjunctively. However, this kind of nesting representation has a limitation because of its tree structure: it is not clear how we can represent three threads t_0, t_1 and t_2 where either e_0 and e_1 succeed or e_0 and e_2 succeed. If we take the idea of event structure [147], we could make a deduction system where each inference step results in a coherence space whose elements are sequents.

6.3.5 Understanding Waitfreedom

Although the definition of waitfreedom is operational (Section 3.3), there is a topological characterization [72, 125] of waitfreely solvable problems using contractibility. Since we have obtained a logical characterization, studying the direct connection between the logical and topological characterizations would be interesting.

6.4 From Philosophical Perspectives

Mathematical logic first succeeded in formalizing mathematics. After that, there have been many attempts to investigate analytic philosophy using formal logics: relevance logic, modal epistemic logics [69], dynamic epistemic logic [140], inquisitive logic [32], deontic logics [143] and so on. Among those investigations, in some cases, substructural logics play important roles. For example, relevance logic is a famous substructural logic lacking weakening and inquisitive logic is a "weak logic" (a logic possibly without substitution-closedness) between intuitionistic and classical logics [32].

Gödel-Dummett logic [43] was invented for algebraic reasons, but now we found a constructive justification of Dummett's axiom $(\varphi \supset \psi) \vee (\psi \supset \varphi)$ as the result of speed-racing during proof reduction. Since the semantics presented in Chapter 3 seems somewhat too complicated, there is room for simplification and better understanding of Gödel-Dummett logic.

Abelian logic can provide a way to express exchanges and indebtedness. The Amida axiom $(\varphi \multimap \psi) \otimes (\psi \multimap \varphi)$ can describe two agents' exchange of φ and ψ between two agents or one agent's borrowing of φ for ψ and returning φ for ψ. Since analysis of pattern of exchange is an important subject of anthropology [97], Abelian logic can provide a basis for describing social and cultural phenomena.

Dynamic epistemic logic [140], or its fragment called public announcement logic, is a modal logic that has a modality for agents' knowledge and announcements. However, in the current formulation of dynamic epistemic logic, we cannot formalize the effect of repetitions[2] of announcements or cancels of announcements. In Abelian logic, we can multiply any formula with any integer, thus there is a possibility of representing repetition or cancels of announcements using a modal logic based on Abelian logic.

Also, since Abelian logic is modeled by Abelian groups, it can express the sense of indebtedness or obligation in a quantitative way. Deontic logic [101, 143] tried to capture permission, obligation and so on. Using Abelian logic, we hope to address how much duty is imposed. Since the Amida calculus can reduce some proofs in Abelian logic, it could provide a way to analyze a complicated situation where many kinds of obligations and permissions of different significance are interwound.

[2]The point is distinguishing a single announcement from repeated announcements of the same content. For that, we do not want contraction.

References

[1] M. Abadi, L. Cardelli, P. L. Curien, and J. J. Levy. Explicit substitutions. In *POPL '90*, pages 31–46, New York, 1990. ACM.

[2] T. Abe. Completeness of modal proofs in first-order predicate logic. *Computer Software*, 24(4):165–177, 2007.

[3] S. Abramsky. Computational interpretations of linear logic. *Theo. Comp. Sci.*, 111(1-2):3–57, 1993.

[4] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In P. Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006.

[5] J. Armstrong. *Programming Erlang: Software for Concurrent World.* The Pragmatic Bookshelf, 2007.

[6] K. Asai and Y. Kameyama. Polymorphic delimited continuations. In *APLAS '07*, volume 4807 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2007.

[7] A. Asperti. and L. Roversi. Intuitionistic light affine logic. *ACM Trans. Comput. Logic*, 3(1):137–175, 1 2002.

[8] A. Avron. Hypersequents, logical consequence and intermediate logics for concurrency. *Ann. Math. Artif. Intell.*, 4:225–248, 1991.

[9] A. Avron. *The method of hypersequents in the proof theory of propositional non-classical logics*, pages 1–32. Clarendon Press, 1996.

[10] A. Avron. A tableau system for Gödel–Dummett logic based on a hypersequent calculus. In *TABLEAUX '00*, volume 1847 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2000.

[11] M. Baaz, A. Ciabattoni, and C. G. Fermüller. A natural deduction system for intuitionistic fuzzy logic. *Lectures on soft computing*, 2001.

[12] M. Baaz, A. Ciabattoni, and C. G. Fermüller. Hypersequent calculi for Gödel logics a survey. *Journal of Logic and Computation*, 13(6):835–861, 2003.

[13] M. Baaz, A. Ciabattoni, and F. Montagna. Analytic calculi for monoidal t-norm based logic. *Fundamenta Informaticae*, 59(4):315–332, 2004.

[14] D. Baelde. Least and greatest fixed points in linear logic. *ACM Trans. Comput. Logic*, 13(1):2:1–2:44, 1 2012.

[15] V. Balat, R. Di Cosmo, and M. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. *SIGPLAN Not.*, 39(1):64–76, 1 2004.

[16] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, 1968. ACM.

[17] A. Bejleri and N. Yoshida. Synchronous multiparty session types. *Electronic Notes in Theoretical Computer Science*, 241(0):3–33, 2009.

[18] G. M. Bierman. A computational interpretation of the $\lambda\mu$-calculus. In L. Brim, J. Gruska, and J. Zlatuka, editors, *Mathematical Foundations of Computer Science 1998*, volume 1450 of *Lecture Notes in Computer Science*, pages 336–345. Springer, 1998.

[19] O. Biran, S. Moran, and S. Zaks. A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. In *PODC '88*, pages 263–275. ACM, 1988.

[20] L. Birkedal, K. Støvring, and J. Thamsborg. Relational parametricity for references and recursive types. In *Proceedings of the 4th international workshop on Types in language design and implementation*, pages 91–104. ACM, 2009.

[21] P. Blackburn. Representation, reasoning, and relational structures: a hybrid logic manifesto. *Logic Journal of IGPL*, 8(3):339–365, 2000.

[22] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *PODC '93*, pages 41–51. ACM, 1993.

[23] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the 12th annual ACM symposium on principles of distributed computing*, PODC '93, pages 41–51. ACM, 1993.

[24] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda–a functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

[25] N. Brown. Combinators for message-passing in Haskell. In R. Rocha and J. Launchbury, editors, *Lecture Notes in Computer Science*, pages 19–33. Springer, 2011.

[26] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In P. Gastin and F. Laroussinie, editors, *CONCUR 2010*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.

[27] E. Casari. Comparative logics and Abelian l-groups. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium '88*, volume 127 of *Studies in logic and the foundations of mathematics*, pages 161–190. North-Holland, 1989.

[28] K. Chvalovský. On the independence of axioms in BL and MTL. *Fuzzy Sets and Systems*, 197(0):123–129, 2012.

[29] A. Ciabattoni. Hypersequent calculi for some intermediate logics with bounded Kripke models. *Journal of Logic and Computation*, 11(2):283–294, 2001.

[30] A. Ciabattoni, N. Galatos, and K. Terui. From axioms to analytic rules in nonclassical logics. In *LICS'08.*, pages 229–240. IEEE, 2008.

[31] Agata Ciabattoni, Lutz Straßburger, and Kazushige Terui. Expanding the realm of systematic proof theory. In Erich Grädel and Reinhard Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2009. ISBN 978-3-642-04026-9.

[32] I. Ciardelli and F. Roelofsen. Inquisitive logic. *Journal of Philosophical Logic*, 40(1):55–94, 2 2011.

[33] P. Cintula, P. Hájek, and R. Horčík. Formal systems of fuzzy logic and their fragments. *Annals of Pure and Applied Logic*, 150(1—3):40–65, 2007.

[34] P. Cintula, P. Hájek, and C. Noguera, editors. *Handbook of Mathematical Fuzzy Logic*. Studies in Logic. College Publications, 2011.

[35] P.-L. Curien and H. Herbelin. The duality of computation. *SIGPLAN Not.*, 35 (9):233–243, 9 2000.

[36] H. B. Curry. The combinatory foundations of mathematical logic. *The Journal of Symbolic Logic*, 7(2):49–64, 1942.

[37] H. B. Curry and R. Feys. *Combinatory logic*, volume 1. North-Holland, third printing edition, 1974.

[38] V. Danos and J.-L. Krivine. Disjunctive tautologies as synchronisation schemes. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic*, volume 1862 of *Lecture Notes in Computer Science*, pages 292–301. Springer, 2000.

[39] V. Danos and L. Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28(3):181–203, 10 1989.

[40] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 151–160. ACM, 1990.

[41] P. de Groote. On the strong normalisation of intuitionistic natural deduction with permutation-conversions. *Information and Computation*, 178(2):441–464, 2002.

[42] C. Doczkal and J. Schwinghammer. Formalizing a strong normalization proof for Moggi's computational metalanguage: a case study in Isabelle/HOL-nominal. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '09, pages 57–63, New York, 2009. ACM.

[43] M. Dummett. A propositional calculus with denumerable matrix. *The Journal of Symbolic Logic*, 24(2):97–106, 1959.

[44] D. Edgington. Conditionals. In Edward N. Z., editor, *The Stanford Encyclopedia of Philosophy*. Winter 2008 edition, 2008.

[45] J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with Craig interpolation and symbolic pushdown systems. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*,

volume 3920 of *Lecture Notes in Computer Science*, pages 489–503. Springer, 2006.

[46] F. Esteva and L. Godo. Monoidal t-norm based logic: towards a logic for left-continuous t-norms. *Fuzzy Sets and Systems*, 124(3):271–288, 2001.

[47] S. Feferman, J. Dawson Jr, Kleene S., G. Moore, R. Solovay, and J. van Heijenoort, editors. *Kurt Gödel Collected Works*, volume I. Oxford University Press, 1986.

[48] C. Fermüller. Parallel dialogue games and hypersequents for intermediate logics. In *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2796 of *Lecture Notes in Computer Science*, pages 48–64. Springer, 2003.

[49] A. Filinski. Declarative continuations: An investigation of duality in programming language semantics. In D. Pitt, D. Rydeheard, P. Dybjer, A. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 224–249. Springer, 1989.

[50] G. Fiorino. Fast decision procedure for propositional Dummett logic based on a multiple premise tableau calculus. *Information Sciences*, 180(19):3633–3646, 2010.

[51] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *POPL '96*, pages 372–385. ACM, 1996.

[52] C. Fournet, L. Maranget, C. Laneve, and D. Rémy. Implicit typing à la ML for the join-calculus. In *CONCUR '97*, volume 1243 of *Lecture Notes in Computer Science*, pages 196–212. Springer, 1997.

[53] C. Fournet, F. L. Fessant, L. Maranget, and A. Schmitt. JoCaml: A language for concurrent distributed and mobile programming. *Advanced Functional Programming*, pages 1948–1948, 2002.

[54] G. Frege. Begriffsschrift. In J. van Heijenoort, editor, *From Frege to Gödel: a source book in mathematical logic, 1879-1931*, pages 1–82. Harvard University Press, 1967. Originally "Begriffsschrift: eine der arithmetischen nachgebildete Formelsprache des reinen Denkens," Halle, 1879.

[55] D. M. Gabbay and R. J. G. B. de Queiroz. Extending the curry-howard interpretation to linear, relevant and other resource logics. *The Journal of Symbolic Logic*, 57(4):1319–1365, 1992.

[56] E. Gafni and E. Koutsoupias. Three-processor tasks are undecidable. *SIAM Journal on Computing*, 28(3):970–983, 1999.

[57] N. Galatos, P. Jipsen, T. Kowalski, and H. Ono. *Residuated Lattices: An Algebraic Glimpse at Substructural Logics*, volume 151 of *Studies in Logic and The Foundations of Mathematics*. Elsevier, 1st edition, 2007.

[58] G. Gentzen. Investigations into logical deduction. *American Philosophical Quarterly*, 1(4):288–306, 1964.

[59] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[60] J.-Y. Girard. *Geometry of Interaction 1: Interpretation of System F*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221–260. Elsevier, 1989.

[61] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[62] M. Giunti and V. T. Vasconcelos. A linear account of session types in the pi calculus. In *CONCUR 2010*, Lecture Notes in Computer Science, pages 432–446. Springer, 2010.

[63] M. V. Glivenko. Sur la logique de M. Brouwer. *Bulletin de la Classe des science, Académie Royale de Belgique*, 5(14):225–228, 1928.

[64] M. V. Glivenko. Sur quelques points de la logique de M. Brouwer. *Bulletin de la Classe des science, Académie Royale de Belgique*, 5(15):185–188, 1929.

[65] K. Gödel. On the intuitionistic propositional calculus. In S. Feferman, J. Dawson Jr., Kleene S., G. Moore, R. Solovay, and J. van Heijenoort, editors, *Kurt Gödel Collected Works*, volume I, pages 222–225. Oxford University Press, 1986. (Translation of "Zum intuitionistischen Aussagenkalkül," *Anzeiger der Akademie der Wissenschaften in Wien*, **69**: pages 65-66. 1932).

[66] T. G. Griffin. A formulae-as-type notion of control. In *POPL '90*, POPL '90, pages 47–58, New York, 1990. ACM.

[67] P. Hájek. On arithmetic in the Cantor-Łukasiewicz fuzzy set theory. *Archive for Mathematical Logic*, 44(6):763–782, 2005.

[68] P. Hájek, Jeff Paris, and John Shepherdson. The liar paradox and fuzzy logic. *The Journal of Symbolic Logic*, 65(1):339–346, 2000.

[69] V. Hendricks and J. Symons. Epistemic logic. In Edward N. Z., editor, *The Stanford Encyclopedia of Philosophy*. Spring 2009 edition, 2009.

[70] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC '88*, pages 276–290. ACM, 1988.

[71] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[72] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46:858–923, 1999.

[73] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann, 1973.

[74] A. Heyting. The formal rules of intuitionistic logic. In Paolo M., editor, *From Brouwer to Hilbert: The Debate on the Foundations of Mathematics in the 1920s*, pages 311–327. Oxford University Press, 1998. Originally "Die formalen Regeln der intuitionistischen Logik," *Sitzungberichte der Preussischen Akademie der Wissenschaften,* 1930, pages 42–56.

[75] Y. Hirai. An intuitionistic epistemic logic for sequential consistency on shared memory. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 272–289. Springer, 2010.

[76] Y. Hirai. An intuitionistic epistemic logic for asynchronous communication. Master's thesis, Department of Computer Science, The University of Tokyo, 2010.

[77] Y. Hirai. A lambda calculus for Gödel-Dummett logic capturing waitfreedom. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming*, volume 7294 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2012.

[78] K. Honda. Types for dyadic interaction. In *CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer.

[79] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 1991.

[80] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 1991.

[81] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL '08*, POPL '08, pages 273–284, New York, 2008. ACM.

[82] T. Hosoi and H. Ono. Intermediate propositional logics (a survey). *Journal of Tsuda College*, 5:67–82, 1973.

[83] K. Imai, S. Yuen, and K. Agusa. Session type inference in Haskell. In Kohei H. and Alan M., editors, *PLACES*, volume 69 of *EPTCS*, pages 74–91, 2010.

[84] M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2000.

[85] Y. Kakutani. Duality between call-by-name recursion and call-by-value iteration. In J. Bradfield, editor, *Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 277–308. Springer, 2002.

[86] A. D. Ker, H. Nickau, and C. H. L. Ong. Innocent game models of untyped l-calculus. *Theoretical Computer Science*, 272(1—2):247–292, 2002.

[87] D. Kimura. *Computation in Classical Logic and Dual Calculus.* PhD thesis, The Graduate University of Advanced Studies, March 2007.

[88] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. Haskell '04, pages 96–107, New York, 2004. ACM.

[89] N. Kobayashi and A. Yonezawa. Higher-order concurrent linear logic programming. In *Theory and Practice of Parallel Programming*, volume 907 of *Lecture Notes in Computer Science*, pages 137–166. Springer, 1995.

[90] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 9 1999.

[91] A. Kolmogorov. On the interpretation of intuitionistic logic. In P. Mancosu, editor, *From Brouwer to Hilbert: The Debate on the Foundations of Mathematics in the 1920s*, pages 328–334. Oxford University Press, 1998. Originally, "Zur Deutung der intuitionistischen Logik," *Mathematische Zeitschrift* **35**, 1932, pages 58–65.

[92] F. Lamarche. Proof nets for intuitionistic linear logic: essential nets. Technical report, Loria & INRIA-Lorraine, 2008.

[93] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, 100(28):690–691, 1979.

[94] P. J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Commun. ACM*, 8(2):89–101, 2 1965.

[95] S. Lindley and I. Stark. Reducibility and TT-lifting for computation types. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications*, volume 3461 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2005.

[96] L. L. Maksimova. Craig's theorem in superintuitionistic logics and amalgamable varieties of pseudo-boolean algebras. *Algebra and Logic*, 16(6):427–455, 1977.

[97] B. Malinowski. Kula; the circulating exchange of valuables in the archipelagoes of eastern New Guinea. *Man*, 20:97–105, 1920.

[98] S. Marlow et al. Haskell 2010 language report. *Available online http://www.haskell.org/onlinereport/haskell2010*, 2010.

[99] S. Martini and A. Masini. A computational interpretation of modal proofs. In H. Wansing, editor, *Proof Theory of Modal Logic*, pages 213–241. Kluwer, 1996.

[100] K. L. McMillan. Interpolation and SAT-based model checking. In Jr. Hunt, W. A. and F. Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[101] P. McNamara. Deontic logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2010 edition, 2010.

[102] G. Metcalfe. Proof theory for Casari's comparative logics. *Journal of Logic and Computation*, 16(4):405–422, 2006.

[103] G. Metcalfe, N. Olivetti, and D. Gabbay. Analytic sequent calculi for Abelian and Łukasiewicz logics. In U. Egly and C. Fermüller, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 2381 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2002.

[104] R. K. Meyer and J. K. Slaney. Abelian logic (from A to Z). In G. Priest, R. Richard, and J. Norman, editors, *Paraconsistent Logic: Essays on the Inconsistent*, chapter IX, pages 245–288. Philosophia Verlag, 1989.

[105] Microsoft. The F# 3.0 language specification. *Available online http://fsharp.org/about/files/spec.pdf*, 2012.

[106] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[107] R. Milner. *Communicating and mobile systems: the pi-calculus.* Cambridge University Press, 1999.

[108] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML.* MIT press, 1997.

[109] Y. Minsky. OCaml for the masses. *Commun. ACM*, 54(11):53–58, 11 2011.

[110] S. Moran. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145–151, 1987.

[111] A. S. Murawski and C. H. L. Ong. Exhausting strategies, joker games and full completeness for IMLL with unit. *Theoretical Computer Science*, 294(1—2): 269–305, 2003.

[112] R. Nakajima. Infinite normal forms for the λ-calculus. In C. Böhm, editor, ***λ****-Calculus and Computer Science Theory*, volume 37 of *Lecture Notes in Computer Science*, pages 62–82. Springer, 1975.

[113] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *PARLE '91*, volume 506 of *Lecture Notes in Computer Science*, pages 202–219. Springer.

[114] A. Ohori and K. Ueno. Making standard ML a practical database programming language. *SIGPLAN Not.*, 46(9):307–319, 9 2011.

[115] C. H. L. Ong and C. A. Stewart. A curry-howard foundation for functional computation with control. In *POPL '97*, POPL '97, pages 215–227, New York, 1997. ACM.

[116] H. Ono and Y. Komori. Logics without the contraction rule. *The Journal of Symbolic Logic*, 50(1):169–201, 1985.

[117] M. Parigot. $\lambda\mu$-Calculus: An algorithmic interpretation of classical natural deduction. In *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992.

[118] M. Parigot. On the computational interpretation of negation. In P. G. Clote and H. Schwichtenberg, editors, *Computer Science Logic*, volume 1862 of *Lecture Notes in Computer Science*, pages 472–484. Springer, 2000.

[119] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almqvist & Wiksell, 1965.

[120] A. N. Prior. The runabout inference-ticket. *Analysis*, 21(2):38–39, Dec 1960.

[121] T. Raths, J. Otten, and C. Kreitz. The ILTP problem library for intuitionistic logic. *Journal of Automated Reasoning*, 38(1):261–271, 2007.

[122] J. Rees and W. Clinger. Revised report on the algorithmic language Scheme. *SIGPLAN Not.*, 21(12):37–79, 12 1986.

[123] J. Reppy. Concurrent ML: Design, application and semantics. In P. Lauer, editor, *Lecture Notes in Computer Science*, pages 165–198. Springer, 1993.

[124] J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

[125] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: the topology of public knowledge. In *STOC '93*, pages 101–110. ACM, 1993.

[126] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: the topology of public knowledge. *SIAM journal on computing*, 29(5):1449–1483, 2000.

[127] P. Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(02): 207–260, 4 2001.

[128] M. Shirahata. A sequent calculus for compact closed categories.

[129] O. Sonobe. A Gentzen-type formulation of some intermediate propositional logics. *Journal of Tsuda College*, 14:7–14, 1975.

[130] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry–Howard isomorphism.* Elsevier, 2007.

[131] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. *PARLE'94 Parallel Architectures and Languages Europe*, pages 398–413.

[132] K. Terui. Light affine lambda calculus and polynomial time strong normalization. *Archive for Mathematical Logic*, 46(3):253–280, 2007.

[133] I. Thomas. Finite limitations on Dummet's LC. *Notre Dame Journal of Formal Logic*, 3(3):170–174, 1962.

[134] T. Toffoli. Reversible computing. In J. Bakker and J. Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 632–644. Springer, 1980.

[135] A. S. Troelstra. *Lectures on Linear Logic.* CSLI, 1992.

[136] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction: Vol. 1.* North-Holland, 1988.

[137] T. Umezawa. On logics intermediate between intuitionistic and classical predicate logic. *The Journal of Symbolic Logic*, 24(2):141–153, 1959.

[138] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP '09*, pages 277–288, New York, 2009. ACM.

[139] A. Urquhart. Semantics for relevant logics. *The Journal of Symbolic Logic*, 37 (1):159–169, 1972.

[140] H. van Ditmarsch, W. van der Hoek, and B. P. Kooi. *Dynamic epistemic logic.* Springer, 2007.

[141] W. P. van Stigt. Brouwer's intuitionist programme. In Paolo M., editor, *From Brouwer to Hilbert*, pages 1–22. Oxford University Press, 1998.

[142] Tom-Murphy VII, K. Crary, and R. Harper. Distributed control flow with classical modal logic. In *CSL'05*, volume 3634 of *Lecture Notes in Computer Science*, pages 51–69. Springer, 2005.

[143] G. H. Von Wright. Deontic logic. *Mind*, 60(237):1–15, 1951.

[144] P. Wadler. Call-by-value is dual to call-by-name. *SIGPLAN Not.*, 38(9):189–201, 8 2003.

[145] P. Wadler. Call-by-value is dual to call-by-name—reloaded. In J. Giesl, editor, *Term Rewriting and Applications 2005*, volume 3467 of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2005.

[146] P. Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN international conference on functional programming*, ICFP '12, pages 273–286, New York, 2012. ACM.

[147] G. Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.

[148] S. Yatabe. Comprehension contradicts to the induction within Łukasiewicz predicate logic. *Archive for Mathematical Logic*, 48(3-4):265–268, 5 2009.

Index