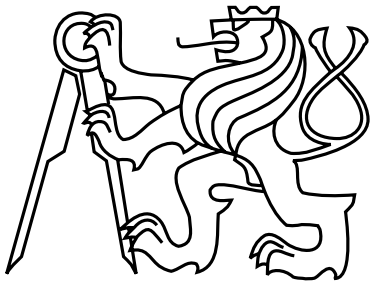
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Bachelor's Thesis

Computational trinitarianism and Linear types

Vojtěch Štěpančík

Supervisor: Ing. Matěj Dostál, Ph.D.

Study Programme: Otevřená informatika, Bakalářský

Field of Study: Software

May 29, 2021

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Štěpančík Vojtěch** Personal ID number: **483531**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Computational trinitarianism and linear types

Bachelor’s thesis title in Czech:

Výpočetní trinitarismus a lineární typy

Guidelines:

Computational trinitarianism describes the intimate relationship between logic, category theory and type theory. This relationship identifies propositions of a logic with a type of a corresponding type system, and also establishes a correspondence between a proof of a proposition, a term (program) of a given type, and a generalised element of an object in a category.
A linear type system is a special kind of a substructural type system with important applications in computer science. An advantage of a linear type system resides in its ability to place constraints on the usage of (or access to) variables (resources).
The aim of the bachelor thesis is to describe linear logic as an example of a substructural logic, to construct a linear type system stemming from that logic, and to give their categorical semantics via categories with structure.
The style and presentation of the thesis will be theoretical.

Bibliography / sources:

[1] G. Restall, An introduction to substructural logics, Routledge, 2000
[2] P. Wadler, A taste of linear logic, In: Borzyszkowski A.M., Sokołowski S. (eds) Mathematical Foundations of Computer Science 1993. MFCS 1993. Lecture Notes in Computer Science, vol 711. Springer, Berlin, Heidelberg. 1993
[3] R. L. Crole, Categories for types, Cambridge University Press, 1994

Name and workplace of bachelor’s thesis supervisor:

Ing. Matěj Dostál, Ph.D., Department of Mathematics, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **04.03.2021** Deadline for bachelor thesis submission: **29.05.2021**

Assignment valid until: **19.02.2023**

Ing. Matěj Dostál, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student's signature

Acknowledgements

I would like to thank my family for their continued support, and my supervisor Matěj Dostál for mentoring me during the writing of this thesis.

I am also very thankful to Jiří Velebil, for his lectures on category theory and feedback on preprints of this work.

I appreciate the discussions with my colleagues Max Hollmann, Matěj Kafka, Jakub Dupák and Jáchym Herynek, which lead to all of us striving to produce the best work we can.

Declaration

I hereby declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on May 28th, 2021 .....

Abstract

This thesis focuses on extending the Curry-Howard correspondence into a linear setting. Instead of the traditional equivalence of intuitionistic logic and simply typed λ-calculus, we present a formulation of linear logic, which corresponds to a language referred to as *linear λ-calculus*. We show the correspondence on three levels — types as propositions, programs as proofs and computation as reduction.

Further, we show an embedding of intuitionistic logic into linear logic, and an analogous embedding of intuitionistic programs into linear programs. The last chapter describes a class of categories with structure, which reflect the behavior of linear programs.

Keywords: natural deduction, linear logic, linear types, type theory, Curry-Howard correspondence, categorical semantics

Abstrakt

Tato práce se zabývá rozšířením Curryovy-Howardovy korespondence do lineárního prostředí. Místo tradiční ekvivalence intuicionistické logiky a jednoduše typovaného λ-kalkulu zavádíme formulaci lineární logiky, která odpovídá jazyku nazývanému *lineární λ-kalkulus*. Korespondenci ukazujeme na třech úrovních — typy jako výroky, programy jako důkazy a výpočet jako redukce.

Dále popisujeme vnoření intuicionistické logiky do lineární logiky, a analogické vnoření intuicionistických programů do lineárních programů. Poslední kapitola se věnuje třídě kategorií se strukturou, které odrážejí chování lineárních programů.

Klíčová slova: přirozená dedukce, lineární logika, lineární typy, teorie typů, Curryova-Howardova korespondence, kategoriální semantika

Contents

List of Figures

Introduction

This thesis explores the idea of *computational trinitarianism*, which presents a correspondence between concepts from logic, type theory and category theory.

The Curry-Howard correspondence gives an isomorphism between propositions of intuitionistic logic and types of the simply typed λ-calculus, and this notion is extended into objects of an appropriate category.

We present an extension of this correspondence, by replacing intuitionistic logic with linear logic. The type system stemming from linear logic has important applications in computer science — it allows the programmer to constrain the usage of variables, representing resources.

Chapter 1 deals with logic. Namely we study fragments of intuitionistic and linear logic, and explain the sense in which the latter can be called substructural. We describe a way in which intuitionistic logic can be embedded into linear logic.

In Chapter 2 we introduce the basics of type theory. Two rudimentary programming languages are introduced: the well-known simply typed λ-calculus and the lesser-known linear λ-calculus. Their relation to the logics introduced in Chapter 1 is explained and an embedding of programs from the simply typed λ-calculus into programs from linear λ-calculus is given.

Further in Chapter 3, we give an overview of category theory, specifically the parts necessary for explaining the semantics of the two languages in terms of categories with structure, and proceed to present the semantics.

Chapter 1

Logic

Mathematical logic is logic treated by mathematical methods. However, such studies of different kinds of logic often use logical and deductive thinking themselves. To separate the logic observed from the logic used to make the observations, we consider them to be two separate systems. The language of the logic studied is referred to as the **object language**, while the language of the logic used for doing the observing is called the **metalanguage** (Kleene, 1966).

These metalanguages are then used to reason about formal composition of proofs — therefore we call them **proof systems**, or **proof calculi**.

The proof system used in this paper stems from Gentzen's natural deduction (Gentzen, 1935). Natural deduction builds proofs on **judgements** and **propositions**.

A proposition is a formula of the object language, and a judgement is a knowable fact. For example in traditional logic (that is to say, a *truth-oriented* logic), one might take "It is raining today" for a proposition A, and a judgement is the statement A *is true*, or A *true* for short.

Another judgment that often arises in various logics is identifying propositions themselves — one can only make judgments about a proposition A if A is a proposition, which is represented by the judgment A *is a proposition*, abbreviated to A *prop*.

We will later see that, without delving into the philosophy of mathematics, the exact nature of propositions and judgements depends on the object language.

The basis for the metalanguage is the **deduction rules**. A deduction rule consists of a collection of judgements, called the **premises**, and a single judgement, called the **conclusion**. To be able to refer to the rule in proofs, it is assigned a semantically significant name. Graphically, it is represented by drawing a horizontal line (the **derivation line**), placing the premises above it, the conclusion below, and writing the name of the rule to the right.

To illustrate, if we wanted to show the rule expressing that given two propositions, A and B, and the judgements A *true* and B *true*, one can obtain the judgement $A \wedge B$ *true*, we could write it as

$$\frac{A \text{ true} \quad A \text{ prop} \quad B \text{ true} \quad B \text{ prop}}{(A \wedge B) \text{ true}} \wedge\text{I}$$

where the label ∧I is an abbreviation for "conjunction introduction".

Gentzen used the concept of assumptions to formulate the rules for implication. If, given that A *true*, we could sequence the deduction rules in such a way that we get the judgement B *true*, we can abstract this dependency on a hypothetical A into an implication. Gentzen used $[A]$ to denote the **assumption** of the judgement A *true*, and this assumption needs to be later **discharged** by abstracting it into an appropriate implication via a corresponding implication introduction. The formulation of the → I rule can be seen in Figure 1.1 (left).

$$\frac{\begin{array}{c}[A]^1\\ \vdots\\ B\end{array}}{A \to B}(\to \mathrm{I})_1 \qquad \frac{A\ prop, A\ true, B\ prop \vdash B\ true}{A\ prop, B\ prop \vdash (A \to B)\ true} \to \mathrm{I}$$

Figure 1.1: Gentzen's assumption notation (left) and notation for localized assumptions (right)

The symbol $\vdots$ stands for a sequence of deduction rules that can derive the judgement B *true* from the judgement A *true*.

A proof in natural deduction is tree-like, with the judgement to be proven at the root, assumptions at the leaves, and deduction rules between the nodes. It is not a proper tree, because it needs to keep track of which implication introductions discharge which assumptions, so additional structure to manage backreferences is necessary.

In this notation, assumptions are *global* to the proof. We can change the notation to be able to reason about assumptions locally, allowing us to degenerate the proof structure to a proper tree. We say that a **contextualized judgement**[1] has the form $\Gamma \vdash J$, where Γ is a sequence of zero or more assumptions, called the **context**, and J is the judgement. An example of rewriting a proof from Gentzen's notation to the context notation is shown in Figure 1.1. Note that the context can also be empty. Assumptions can be added to the context via a comma: Γ, S is a new context, which includes all the assumptions from Γ, and the assumption S. This concatenation is extended in the obvious way to merging of two contexts, so Γ, Δ is a context that includes all the assumptions from Γ, and all the assumptions from Δ (Pfenning, 2004). In this new notation, deduction rules have contextualized judgements for premises and conclusion.

The behaviour of the context is specified in the metalanguage, using deduction rules. These rules, acting on the context, are called **structural rules**, and usually include Weakening, Contraction, and Exchange, which are listed in Figure 1.2. These three rules encode semantics similar to those of a finite set.

Weakening allows one to add arbitrary assumptions to the context without invalidating the derived judgement. Contraction states that assumptions may be used multiple times. Exchange asserts that the order in which assumptions appear in the context is irrelevant.

A logic which constrains one or more of these structural rules is called **substructural** (Paoli, 2013).

[1] The notation is borrowed from Gentzen's other proof calculus, the sequent calculus. To prevent confusion of the two systems, we prefer the term *contextualized judgment* to Gentzen's *sequent*.

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

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

$$\frac{\Gamma, A\ true, B\ true, \Delta \vdash C\ true}{\Gamma, B\ true, A\ true, \Delta \vdash C\ true}\ \text{Exchange}$$

Figure 1.2: Structural rules

Apart from the structural rules, the logic also specifies **logical rules**. These describe how the logical connectives participate in derivations. Conventionally, they come in pairs of introduction and elimination, the former defining how a proposition containing the connective is created, and the latter defining how such a proposition is "used" and split apart.

Just as there can be zero assumptions in a contextualized judgement, there can be zero premises in a deduction rule. Such rules are called **axioms**, and the judgments in their conclusions are always derivable.

A proof in this updated notation is now a proper tree, with a contextualized judgement at the root, contextualized judgements in the inner nodes, axioms at the leaves, and deduction rules connecting the nodes.

When composing deductions, we sometimes produce proofs with redundancies. Namely when a rule for introducing a connective is immediately followed by a rule for eliminating it, the proof can be simplified via rewriting rules called **proof-reductions**. These rules must preserve well-formedness of the proof, meaning that the proof after a reduction must still consist only of derivations specified for the logic. This condition is called *local soundness* (Pfenning, 2004).

1.1 Intuitionistic logic

Intuitionistic logic is the logic of constructive mathematics — the only axiom in the system is $A\ true \vdash A\ true$, in other words, any judgement can be made assuming itself. This is in contrast with classical logic, which also axiomatizes the law of excluded middle, $\vdash (A \vee \neg A)\ true$. The philosophical difference between classical and intuitionistic logic is that classical logic is content with knowing whether a proposition is true or whether it is false. After all, those are the only options. Intuitionistic logic, on the other hand, requires a constructive proof — a "recipe", turning the assumptions into the conclusion. The law of excluded middle allows for proofs where one judges a proposition to be true, just because it cannot be false. This goes against the intuitionistic line of reasoning, because merely showing that something has to exists does not provide the mathematician with a way to construct it. In intuitionistic logic, the judgement $(A \vee \neg A)\ true$ can still be made, but it needs to be accompanied with either a proof of $A\ true$ or $\neg A\ true$ (Sørensen & Urzyczyn, 2006).

$$\frac{}{A \vdash_T A}\,\mathrm{Id} \qquad \frac{\Gamma \vdash_T A}{\Gamma, B \vdash_T A}\,\mathrm{Weakening}$$

$$\frac{\Gamma, A, A \vdash_T B}{\Gamma, A \vdash_T B}\,\mathrm{Contraction} \qquad \frac{\Gamma, A, B, \Delta \vdash_T C}{\Gamma, B, A, \Delta \vdash_T C}\,\mathrm{Exchange}$$

$$\frac{\Gamma \vdash_T A \qquad \Gamma \vdash_T B}{\Gamma \vdash_T A \wedge B}\,\wedge\mathrm{I}$$

$$\frac{\Gamma \vdash_T A \wedge B}{\Gamma \vdash_T A}\,\wedge\mathrm{E}_1 \qquad \frac{\Gamma \vdash_T A \wedge B}{\Gamma \vdash_T B}\,\wedge\mathrm{E}_2$$

$$\frac{\Gamma, A \vdash_T B}{\Gamma \vdash_T A \to B}\,\to\mathrm{I} \qquad \frac{\Gamma \vdash_T A \to B \qquad \Delta \vdash_T A}{\Gamma, \Delta \vdash_T B}\,\to\mathrm{E}$$

Figure 1.3: Deduction rules for the meet-implicative fragment of propositional intuitionistic logic

Since intuitionistic logic is an example of a traditional logic, the basic judgement that can be made about a proposition stays the same, *A true*. Because this is the only judgment we will be using in the proofs[2], we define a shorthand notation, $\Gamma \vdash_T A$, where Γ is a list of *propositions*, and A is a proposition, and we take it to mean the contextualized judgment where the context is a list of judgments *P true* for every proposition P in Γ, and where the conclusion is the judgment *A true* (the index T stands for "truth"). For example, the formula $A, B \vdash_T C$ is short for *A true*, *B true* $\vdash$ *C true*. This notation will be used exclusively in the diagrams to prevent them from spreading too wide, and we will use the full form in the body of the thesis.

The logic studied in this section is the meet-implicative fragment of propositional intuitionistic logic — that is to say, we only concern ourselves with propositions created using the connectives $\wedge$ and $\to$. The propositions of this fragment can be described by the following Backus-Naur form:

$$A, B ::= X \mid (A \to B) \mid (A \wedge B)$$

for X ranging over atomic propositions. The rules of this fragment are given in Figure 1.3.

The rules consist of the one axiom Id mentioned above, the three structural rules, Weakening, Contraction, and Exchange, and introduction and elimination rules for the two connectives, $\wedge$I, $\wedge\mathrm{E}_1$, $\wedge\mathrm{E}_2$, $\to$ I and $\to$ E.

Conjunction introduction, labeled $\wedge$I in the deduction rules, states that given a proof of *A true* and a proof of *B true*, the two proofs combined give a proof of *(A* $\wedge$ *B) true*. The respective elimination rules allow one to extract one of the proofs of *A true* or *B true* from *(A* $\wedge$ *B) true*, even after they were combined.

[2]The judgment *A prop* (and subsequently *A type*) is used more frequently in predicate logic and dependent type theories, which are out of scope for this thesis. The well-formed propositions of the relevant fragment can be described more easily with a simple grammar.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ s\end{array} \qquad \begin{array}{c}\vdots\\ t\end{array}}{\dfrac{\Gamma \vdash_T A \qquad \Gamma \vdash_T B}{\Gamma \vdash_T A \wedge B}\wedge\mathrm{I}}\wedge\mathrm{E}_1}{\Gamma \vdash_T A} \quad \Rightarrow \quad \begin{array}{c}\vdots\\ s\end{array}\\ \Gamma \vdash_T A$$

Figure 1.4: Conjunction proof reduction

When formulating the proof reduction rule for a particular connective, one needs to look at a generic example of a reducible proof. For sequencing a conjunction introduction and a conjunction elimination, we need to represent generic proofs of the premises, then apply the two rules in succession, and finally justify an alternative path to reach the conclusion. We can represent the generic proofs with the symbol $\vdots$, much like how Gentzen formulated assumptions. For the conjunction reduction, the generic schema looks like the following tree, with the subproofs labeled s and t.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ s\end{array} \qquad \begin{array}{c}\vdots\\ t\end{array}}{\dfrac{\Gamma \vdash_T A \qquad \Gamma \vdash_T B}{\Gamma \vdash_T A \wedge B}\wedge\mathrm{I}}\wedge\mathrm{E}_1}{\Gamma \vdash_T A}$$

It is easy to see that the conclusion $\Gamma \vdash A\ \mathit{true}$ could have been reached earlier with the s subproof. The full rule is shown in Figure 1.4. The rule for the other elimination rule is not shown, as it is trivially symmetrical.

Implication introduction, labeled $\rightarrow$ I, once again builds on abstracting away an assumption. If a judgement $B\ \mathit{true}$ can be made under an assumption $A\ \mathit{true}$, then the proof tree can be seen as a way of turning a proof of $A\ \mathit{true}$ (or multiple proofs of $A\ \mathit{true}$) into a proof of $B\ \mathit{true}$. The implication elimination is then a method for providing such a proof of A.

The proof reduction rule must take into account that the judgment $A\ \mathit{true}$ might have been assumed zero or multiple times in the proof of $B\ \mathit{true}$, and the context later modified with contractions or weakenings to reach the contextualized judgment $\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}$. Every assumption of $A\ \mathit{true}$ that is used in the proof must have been introduced by the identity rule, and the ones that are not used were introduced by weakening. As shown in (Wadler, 1993), applications of structural and logic rules commute, so for every proof where contraction and weakening are used, there is an equivalent proof with all the contractions and weakenings pushed to the root of the proof tree. In other words, for every proof of $\Gamma, J_1 \vdash J_2$, where J_1 and J_2 stand for arbitrary judgments, there is an equivalent proof which consists of a contraction- and weakening-less subproof of $\Gamma, J_1 \cdots \vdash J_2$, followed by applications of contraction and weakening to accommodate the context, where the ellipsis indicates zero or more assumptions of J_1. The final applications of contraction and weakening are represented by a doubled derivation line, to indicate that it's multiple steps shown as one.

The role of the proof reduction is then to take the proof of $\Delta \vdash A\ \mathit{true}$, and replace with it the instances of $A\ \mathit{true} \vdash A\ \mathit{true}$ in the proof of $\Gamma, A\ \mathit{true} \vdash B\ \mathit{true}$. The full proof reduction rule is shown in Figure 1.5.

$$\dfrac{\dfrac{\dfrac{\left(\dfrac{}{A \vdash_T A}\,\mathrm{Id}\right)\cdots}{\vdots\ s}}{\dfrac{\Gamma, A \cdots \vdash_T B}{\Gamma, A \vdash_T B}\to \mathrm{I}} \qquad \dfrac{\vdots\ t}{\Delta \vdash_T A}}{\Gamma, \Delta \vdash_T B}\to \mathrm{E} \quad \Rightarrow \quad \dfrac{\left(\begin{array}{c}\vdots\ t\\ \Delta \vdash_T A\end{array}\right)\cdots}{\begin{array}{c}\vdots\ s\\ \Gamma, \Delta \cdots \vdash_T B\end{array}}{\Gamma, \Delta \vdash_T B}$$

Figure 1.5: Implication proof reduction

$$\dfrac{\dfrac{\dfrac{}{A \vdash_T A}\,\mathrm{Id} \quad \dfrac{}{A \vdash_T A}\,\mathrm{Id}}{A, A \vdash_T A \wedge A}\wedge \mathrm{I}}{A \vdash_T A \wedge A}\,\mathrm{Contr}}{\vdash_T A \to (A \wedge A)}\to \mathrm{I} \qquad \dfrac{\dfrac{\dfrac{}{A \vdash_T A}\,\mathrm{Id}}{A, B \vdash_T A}\,\mathrm{Weak}}{A \vdash_T B \to A}\to \mathrm{I}}{\vdash_T A \to (B \to A)}\to \mathrm{I}$$

Figure 1.6: Duplication and discard of truth

1.2 Linear logic

In contrast to intuitionistic logic, linear logic considers propositions to be a form of resource — they should not be subject to duplication or discard. When looking at intuitionistic proofs, such as the ones listed in Figure 1.6, we can see that intuitionistic logic has no problem with duplicating propositions (from a single A one might obtain multiple A's) or discarding propositions (the B is unnecessary in the proof of A, so it is thrown away).

In intuitionistic logic, we judged a proposition to be true, and the judgment had the form A *true*. In linear logic, we focus on *availability*. We can judge a proposition A to be available, written A *avail*, if there is a proof that "consumes" some assumptions, "producing" the proposition A. The semantics of consumption are embedded in the deduction rules, explained below.

One simple way to prevent "invalid" usage of resources is to remove the contraction and weakening rules altogether. However, this approach severely limits the expressivity of the language. We might still want to model "free" resources, meaning resources that can be used any number of times, even zero, but conveying this information would not be possible in such a system. Instead, we introduce an annotation for unbound resources, and limit contraction and weakening so that they can only be used on these "intuitionistic" resources. This alternative gives us strictly greater expressivity than intuitionistic logic, as we will see that every intuitionistic proof can be translated to an equivalent linear proof.

The introduction of unbound resources necessitates differentiating between two kinds of assumptions in contextualized judgments — a *linear* assumption of the judgment A *avail* is written $\langle A\ \mathit{avail}\rangle$, and indicates that the conclusion uses the fact that A is available *exactly once*. An *intuitionistic* assumption of the judgment A *avail*, written $[A\ \mathit{avail}]$, makes no guarantees about its usage in the conclusion — it may be used zero, one, or even more times. It is important to emphasize that these glyphs are not a part of the object language

— neither $\langle A \rangle$ nor $[A]$ are well-formed propositions, and the bracket notation can only appear on the left side of a turnstile.

Contraction and weakening are now limited to only intuitionistic assumptions, meaning that judgments can be linearly assumed multiple times. These new rules lead to a general context Γ behaving like a multiset. Every intuitionistic judgment can be made to have a multiplicity of one (using the new contraction and weakening), and multiplicity of linear assumptions is given by their usage in the conclusion.

Similarly to the intuitionistic case, a shorthand notation for contextualized judgments is used — writing $\Gamma \vdash_R A$, the context Γ is a list of *propositions* in square or angle brackets, such as $\langle B \rangle$ or $[C \multimap D]$, and A is a proposition (the index R indicates that we make judgments about resources). This is shorthand for a contextualized judgment whose context is a list containing one occurrence of the judgment $\langle B\ \mathit{avail} \rangle$ for every proposition B in angle brackets in Γ, and one occurrence of the judgment $[C\ \mathit{avail}]$ for every proposition C in square brackets in Γ. The conclusion of this contextualized judgment is the judgment $A\ \mathit{avail}$, where A is the proposition on the right of the turnstile in the shorthand.

A general context Γ can contain assumptions of both kinds, linear and intuitionistic, but an *intuitionistic context*, denoted by $[\Gamma]$, is a context that only contains intuitionistic assumptions, if any.

The focus of this chapter is a fragment of propositional intuitionistic linear logic. It bears similarity to the intuitionistic logic described in the last chapter, specifically it provides tools for representing implication and conjunction, in addition to the linear-logic-specific exponentiation.

The new implication connective is historically called "lollipop", and it's written $A \multimap B$. The proposition is read "produce B consuming A".

Interestingly, there are two conjunction connectives — the "tensor", written $A \otimes B$, and the "with", written $A \,\&\, B$. The tensor represents a conjunction "containing" *both* resources A and B, while the "with" lists two resources that are both available, but not at the same time — the recipient of such a resource needs to choose either A or B.

The last connective is a new concept entirely. The exponential operator $!A$, pronounced "bang", allows one to represent an infinite amount of a resource. We will see how this connective differs from the intuitionistic assumption $[A\ \mathit{avail}]$ and why they are both necessary once we take a look at program evaluation in Part III.

The propositions of this logic can also be described by the simple grammar

$$A, B ::= X \mid (A \multimap B) \mid (A \otimes B) \mid (A \,\&\, B) \ \mid\ !A$$

for X ranging over atomic propositions. The deduction rules are listed in Figure 1.7.

There are now two axioms, one for each kind of assumption. The *linear identity* $\langle \mathrm{Id} \rangle$ says that one can conclude the availability of a resource if one such resource is available. The *intuitionistic identity* expresses the very same concept, except with one caveat — the proof says nothing about how many times the resource was used in the reasoning.

The exchange rule stays unchanged, only S and T stand for any two propositions with square or angle brackets — we are free to rearrange and intermix linear and intuitionistic assumptions.

$$\frac{}{\langle A\rangle \vdash_R A}\,\langle\mathrm{Id}\rangle \qquad \frac{}{[A] \vdash_R A}\,[\mathrm{Id}]$$

$$\frac{\Gamma, S, T, \Delta \vdash_R A}{\Gamma, T, S, \Delta \vdash_R A}\,\mathrm{Exchange}$$

$$\frac{\Gamma, [A], [A] \vdash_R B}{\Gamma, [A] \vdash_R B}\,\mathrm{Contraction} \qquad \frac{\Gamma \vdash_R B}{\Gamma, [A] \vdash_R B}\,\mathrm{Weakening}$$

$$\frac{\Gamma, \langle A\rangle \vdash_R B}{\Gamma \vdash_R (A \multimap B)}\,\multimap \mathrm{I} \qquad \frac{\Gamma \vdash_R (A \multimap B) \quad \Delta \vdash_R A}{\Gamma, \Delta \vdash_R B}\,\multimap \mathrm{E}$$

$$\frac{\Gamma \vdash_R A \quad \Gamma \vdash_R B}{\Gamma \vdash_R A \,\&\, B}\,\&\mathrm{I}$$

$$\frac{\Gamma \vdash_R A \,\&\, B}{\Gamma \vdash_R A}\,\&\mathrm{E}_1 \qquad \frac{\Gamma \vdash_R A \,\&\, B}{\Gamma \vdash_R B}\,\&\mathrm{E}_2$$

$$\frac{\Gamma \vdash_R A \quad \Delta \vdash_R B}{\Gamma, \Delta \vdash_R A \otimes B}\,\otimes \mathrm{I} \qquad \frac{\Gamma, \langle A\rangle, \langle B\rangle \vdash_R C \quad \Delta \vdash_R A \otimes B}{\Gamma, \Delta \vdash_R C}\,\otimes \mathrm{E}$$

$$\frac{[\Gamma] \vdash_R A}{[\Gamma] \vdash_R\, !A}\,!\mathrm{I} \qquad \frac{\Gamma, [A] \vdash_R B \quad \Delta \vdash_R\, !A}{\Gamma, \Delta \vdash_R B}\,!\mathrm{E}$$

Figure 1.7: Deduction rules for the fragment of intuitionistic linear logic

$$\dfrac{\dfrac{\dfrac{}{\langle A\rangle \vdash_R A}\langle\mathrm{Id}\rangle}{\vdots\ s}}{\dfrac{\dfrac{\Gamma, \langle A\rangle \vdash_R B}{\Gamma \vdash_R A \multimap B}\multimap \mathrm{I} \qquad \vdots\ t}{\Gamma, \Delta \vdash_R B}\multimap \mathrm{E}} \quad \Rightarrow \quad \dfrac{\vdots\ t}{\dfrac{\Delta \vdash_R A}{\vdots\ s}}{\Gamma, \Delta \vdash_R B}$$

Figure 1.8: Lollipop proof reduction

$$\dfrac{\dfrac{\vdots\ s \qquad \vdots\ t}{\dfrac{\Gamma \vdash_R A \qquad \Gamma \vdash_R B}{\Gamma \vdash_R A \,\&\, B}\,\&\mathrm{I}}\,\&\mathrm{E}_1}{\Gamma \vdash_R A} \quad \Rightarrow \quad \dfrac{\vdots\ s}{\Gamma \vdash_R A}$$

Figure 1.9: With conjunction proof reduction

The contraction and weakening rules are limited to intuitionistic assumptions, as mentioned in the introduction.

The $\multimap$ ("lollipop") introduction rule in linear logic also abstracts an assumption, but it is limited only to linear ones. The proposition $A \multimap B$ represents an action of "consuming" a resource A to "produce" a resource B. We choose the word "consuming", because when introducing the lollipop, the resource A is removed from the context. In other words, the subsequent deductions loose access to it. Because the deduction sequence leading to the judgment B *avail* was using the assumption $\langle A\ \mathit{avail}\rangle$, we can imagine a proof of the judgment $(A \multimap B)$ *avail* to contain a hole, waiting for an A.

The corresponding elimination rule fills such a hole with a resource obtained from a different context. Emphasis is put on the contexts being different — the context Γ contains other resources that are also consumed during the process of turning an A into a B, therefore the resources cannot be shared with the context used for filling the hole.

Proof reduction for the lollipop is similar in spirit to the intuitionistic implication, except there is no need to worry about the assumption A *avail* being used multiple times. This is apparent from the fact that linear assumptions cannot be contracted. Therefore, the resulting reduction rule is simpler, as shown in Figure 1.8.

The & ("with") deduction rules exactly mirror the intuitionistic conjunction rules. This connective is also called the *additive conjunction*, because the introduction rule shares the resources used for producing the individual components. This sharing of resources prevents a consumer from extracting both of the components — the resources are all used once one of the components is extracted. The proof reduction is also analogous, and presented in Figure 1.9.

The $\otimes$ ("tensor") conjunction represents a pair of resources, both of which have to be consumed, due to the requirement of not discarding resources. The introduction rule looks almost exactly the same as the one for the & conjunction, however in this case, the two parts of the tensor conjunction are produced in different contexts. It is this difference that makes

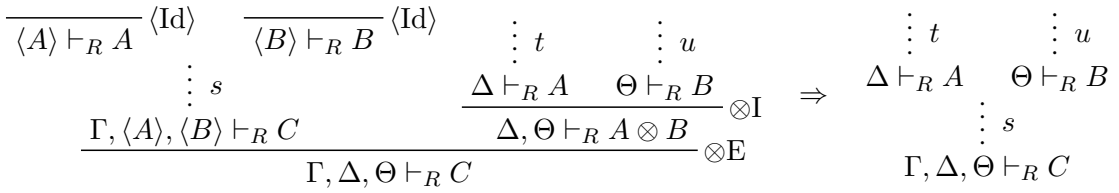


Figure 1.10: Tensor conjunction proof reduction

the two connectives have different semantics — while the & conjunction offers two different possible results from the same resources, the ⊗ conjunction combines two sets of resources into a pair of two results, and provides both for later consumption.

The elimination rule says that a ⊗ resource can be used to complete a proof that contains a linear assumption of each of its constituents. In other words, to consume a ⊗ resource, one must consume both of its parts.

The reduction rule, shown in Figure 1.10, describes how to perform such a completion. If the conjunction is constructed using two proofs t and u of the judgments A *avail* and B *avail*, respectively, then these proofs can replace the assumptions $\langle A\ \mathit{avail}\rangle$ and $\langle B\ \mathit{avail}\rangle$ in another proof s.

The ! ("bang") connective is supposed to extend the expressive power of linear logic to reason about free resources. A judgment of the form $!A$ *avail* does not represent an instance of the resource A, but rather *a source of*[3] these resources. The idea is that a resource A can be pulled out from this source at any time, or even never at all, allowing us to model free resources — the judgment $!A$ *avail* serves as a statement that A is a free resource.

To produce one of these sources, the introduction rule provides us with a way of extending proofs based on only intuitionistic assumptions. Intuitionistic assumptions are another way of modeling free resources, so the essence of the introduction rule is an observation that, given a recipe of creating one unit of a resource A from free ingredients $[\Gamma]$, we can duplicate those free ingredients however many times is necessary to supply more instances of the resource, and that we do not mind throwing the ingredients away in the case that there is no demand for it.

Dually to the introduction rule, which relays how to create a source from free ingredients, the elimination rule describes how a source can satiate an undisclosed demand. A proof built on an intuitionistic assumption gives no guarantees about the number of times it uses the associated resource A. To satisfy this assumption, we can provide the proof with a source $!A$, which can adapt to its requirements.

Reducing a sequence of ! introduction and elimination looks similar to reducing an implication in intuitionistic logic, because it operates on the same principle — replacing assumptions with auxiliary proofs, while acknowledging the fact that the assumptions might appear zero or more times. In the Figure 1.11, the expression $[A]\cdots$ represents zero or more intuitionistic assumptions of the judgment A *avail*, and the proof tree s is devoid of contraction and weakening on the judgment A *avail*. Instead, these are all applied in the step

[3] Or *a generator of*, or *an infinite pocket of*

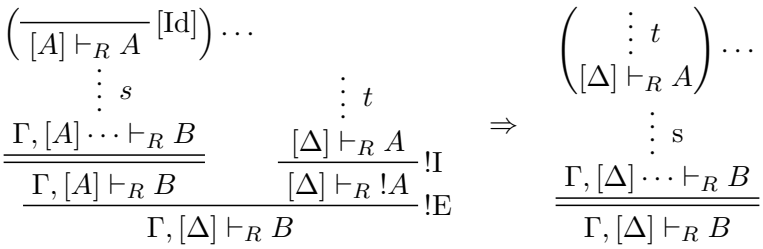


Figure 1.11: Exponential proof reduction

represented by the double derivation line. The reduction then replaces each instance of the intuitionistic assumption A *avail* with a derivation tree t, which produces a resource A from other intuitionistic assumptions. The double line in the reduced proof signifies applications of contraction and weakening to the assumptions $[\Delta]$, corresponding to the double line in the non-reduced proof.

1.3 Intuitionistic embedding

We claimed that every intuitionistic proof can be translated to an equivalent linear proof. To verify this statement, two steps are necessary. First, we need to show how to translate the three primitive constructs: propositions, judgments, and contextualized judgments. Secondly, we need to show that this translation preserves deduction rules and proof reductions. That is to say, for every intuitionistic deduction rule or proof reduction, there is a corresponding linear deduction or reduction taking the translated premises to the translated conclusion.

The intuitionistic propositions come in three flavors: base propositions, conjunctions and implications. We define a translation operator $[\![\_]\!]_L$, and its action on propositions is given by the equations, following (Wadler, 1993),

$$
\begin{aligned}
[\![X]\!]_L &= X \\
[\![A \wedge B]\!]_L &= [\![A]\!]_L \,\&\, [\![B]\!]_L \\
[\![A \to B]\!]_L &= \,!\,[\![A]\!]_L \multimap [\![B]\!]_L
\end{aligned}
$$

where X stands for an atomic proposition, and A and B stand for arbitrary intuitionistic propositions.

On a formal level, this mapping is justified by showing that it preserves deduction and reduction, which is done later in the chapter. On an intuitive level, we appeal to the interpretation of the connectives. When looking at an atomic proposition in isolation, the intuitionistic and linear interpretation is the same, because differences arise only when talking about more complex propositions, and how they relate to each other, for example how are the two sides of a conjunction used, or how is the input to an implication used. The intuitionistic conjunction gives access to each of its constituents, but only one can be extracted, behaving the same as the & conjunction. Finally, the intuitionistic implication gives no

guarantees about the use of its hypothesis, therefore it is necessary to mark the hypothesis with a bang, and promote it to a source in the linear interpretation.

There are only two judgments in intuitionistic logic, and these are $A\ prop$ and $A\ true$ for an intuitionistic proposition A. These are interpreted as $A\ prop$ and $A\ avail$, respectively, defining the action of the translation operator on judgments.

$$\begin{aligned}\llbracket A\ prop \rrbracket_L &= A\ prop \\ \llbracket A\ true \rrbracket_L &= A\ avail\end{aligned}$$

To give a translation of a contextualized judgment, we need to describe how to translate the context. This action is defined with an equation for the empty context, labeled '$\cdot$', and an equation for a concatenation of an arbitrary context Γ with an arbitrary assumption J.

$$\begin{aligned}\llbracket \cdot \rrbracket_L &= \cdot \\ \llbracket \Gamma, J \rrbracket_L &= \llbracket \Gamma \rrbracket_L , [\llbracket J \rrbracket_L]\end{aligned}$$

Verbally, the translation preserves the empty context, and it maps every judgment J in Γ (since assumptions in intuitionistic logic are simply judgments) to an intuitionistic assumption of the translation of the judgment. As a consequence, all the assumptions in a translated context are intuitionistic. The contextualized judgment translation is then given by the equation

$$\llbracket \Gamma \vdash J \rrbracket_L = \llbracket \Gamma \rrbracket_L \vdash \llbracket J \rrbracket_L$$

It is easy to see that by also defining the action of the translation on lists of propositions as $\llbracket (\Gamma_i)_{i=0}^n \rrbracket_L = ([\llbracket \Gamma_i \rrbracket_L])_{i=0}^n$, we can recover a relationship between the shorthand notations:

$$\llbracket \Gamma \vdash_T A \rrbracket_L = \llbracket \Gamma \rrbracket_L \vdash_R \llbracket A \rrbracket_L$$

Having defined the translation of contextualized judgments, we continue by defining how their relationships are translated — that is, how to translate deduction rules.

The axiom of intuitionistic logic is translated into the intuitionistic axiom of linear logic, and the structural rules correspond to their respective counterparts, as shown in the following equations:

$$\begin{aligned}
\left\llbracket \frac{}{A \vdash_T A}\,\mathrm{Id} \right\rrbracket_L &= \frac{}{[\llbracket A \rrbracket_L] \vdash_R \llbracket A \rrbracket_L}\,[\mathrm{Id}] \\
\left\llbracket \frac{\Gamma \vdash_T A}{\Gamma, B \vdash_T A}\,\mathrm{Weakening} \right\rrbracket_L &= \frac{\llbracket \Gamma \rrbracket_L \vdash_R \llbracket A \rrbracket_L}{\llbracket \Gamma \rrbracket_L , [\llbracket B \rrbracket_L] \vdash_R \llbracket A \rrbracket_L}\,\mathrm{Weakening} \\
\left\llbracket \frac{\Gamma, A, A \vdash_T B}{\Gamma, A \vdash_T B}\,\mathrm{Contraction} \right\rrbracket_L &= \frac{\llbracket \Gamma \rrbracket_L , [\llbracket A \rrbracket_L], [\llbracket A \rrbracket_L] \vdash_R \llbracket B \rrbracket_L}{\llbracket \Gamma \rrbracket_L , [\llbracket A \rrbracket_L] \vdash_R \llbracket B \rrbracket_L}\,\mathrm{Contraction} \\
\left\llbracket \frac{\Gamma, A, B, \Delta \vdash_T C}{\Gamma, B, A, \Delta \vdash_T C}\,\mathrm{Exchange} \right\rrbracket_L &= \frac{\llbracket \Gamma \rrbracket_L , [\llbracket A \rrbracket_L], [\llbracket B \rrbracket_L], \llbracket \Delta \rrbracket_L \vdash_R \llbracket C \rrbracket_L}{\llbracket \Gamma \rrbracket_L , [\llbracket B \rrbracket_L], [\llbracket A \rrbracket_L], \llbracket \Delta \rrbracket_L \vdash_R \llbracket C \rrbracket_L}\,\mathrm{Exchange}
\end{aligned}$$

Translation of the intuitionistic conjunction is defined in terms of the & conjunction, so it is expected that the deduction rules of one will correspond to the deduction rules of the other. That is indeed the case, as the translation is given below. It uses the equality $[\![A \wedge B]\!]_L = [\![A]\!]_L \,\&\, [\![B]\!]_L$.

$$\left[\!\!\left[\frac{\Gamma \vdash_T A \quad \Gamma \vdash_T B}{\Gamma \vdash_T A \wedge B} \wedge\mathrm{I} \right]\!\!\right]_L = \frac{[\![\Gamma]\!]_L \vdash_R [\![A]\!]_L \quad [\![\Gamma]\!]_L \vdash_R [\![B]\!]_L}{[\![\Gamma]\!]_L \vdash_R [\![A]\!]_L \,\&\, [\![B]\!]_L} \,\&\mathrm{I}$$

$$\left[\!\!\left[\frac{\Gamma \vdash_T A \wedge B}{\Gamma \vdash_T A} \wedge\mathrm{E}_1 \right]\!\!\right]_L = \frac{[\![\Gamma]\!]_L \vdash_R [\![A]\!]_L \,\&\, [\![B]\!]_L}{[\![\Gamma]\!]_L \vdash_R [\![A]\!]_L} \,\&\mathrm{E}_1$$

The intuitionistic implication is translated with the ! and $\multimap$ connectives, and the translation of the $\to$ I rule, stated below, demonstrates why. The linear implication cannot be introduced from an intuitionistic assumption, so it necessitates an intermediary step which replaces it with a linear assumption, through ! elimination.

$$\left[\!\!\left[\frac{\Gamma, A \vdash_T B}{\Gamma \vdash_T A \to B} \to\mathrm{I} \right]\!\!\right]_L = \frac{\dfrac{[\![\Gamma]\!]_L, [[\![A]\!]_L] \vdash_R [\![B]\!]_L \quad \overline{\langle ![\![A]\!]_L \rangle \vdash_R ![\![A]\!]_L}\,\langle\mathrm{Id}\rangle}{[\![\Gamma]\!]_L, \langle ![\![A]\!]_L \rangle \vdash_T [\![B]\!]_L}\,!\mathrm{E}}{[\![\Gamma]\!]_L \vdash_R ![\![A]\!]_L \multimap [\![B]\!]_L} \multimap\mathrm{I}$$

The translation for the $\to$ E rule takes advantage of the fact that for any intuitionistic context Δ, its translation $[\![\Delta]\!]_L$ consists only of intuitionistic assumptions, therefore it is a valid target for applying the !I rule. Producing a ! proposition is required for the input of the translated implication proposition.

$$\left[\!\!\left[\frac{\Gamma \vdash_T A \to B \quad \Delta \vdash_T A}{\Gamma, \Delta \vdash_T B} \to\mathrm{E} \right]\!\!\right]_L = \frac{[\![\Gamma]\!]_L \vdash_R ![\![A]\!]_L \multimap [\![B]\!]_L \quad \dfrac{[\![\Delta]\!]_L \vdash_R [\![A]\!]_L}{[\![\Delta]\!]_L \vdash_R ![\![A]\!]_L}\,!\mathrm{I}}{[\![\Gamma]\!]_L, [\![\Delta]\!]_L \vdash_R [\![B]\!]_L} \multimap\mathrm{E}$$

We can extend the notion of translating deduction rules into translating entire proof trees. The linear translation of an intuitionistic proof tree p is denoted $[\![p]\!]_L$, and it is constructed by replacing the intuitionistic contextualized judgments and deduction rules by their linear translations. Because the deduction rules are translated into well-formed linear deductions, and because the premises and conclusions are consistently translated, we can be certain that the new deduction tree is correctly constructed and represents a well-formed linear proof.

Finally, we need to show that the translation commutes with reductions. That is, given an intuitionistic proof p and its reduction $p \Rightarrow p'$, there is an equivalent reduction $[\![p]\!]_L \, [\![\Rightarrow]\!]_L \, [\![p]\!]'_L$ such that its result is the same as translating p'. This condition is represented by the following diagram:

$$\begin{array}{ccc} p & \xmapsto{\quad\Rightarrow\quad} & p' \\ \Big\downarrow{\scriptstyle [\![\_]\!]_L} & & \Big\downarrow{\scriptstyle [\![\_]\!]_L} \\ [\![p]\!]_L & \xmapsto{\;[\![\Rightarrow]\!]_L\;} & [\![p]\!]'_L = [\![p']\!]_L \end{array}$$

To prove this commutativity, it suffices to prove it for the two intuitionistic reductions individually.

For conjunction reduction, we take a general reducible proof p

$$p \quad = \quad \dfrac{\dfrac{\begin{array}{cc}\vdots\, s & \vdots\, t \\ \Gamma \vdash_T A & \Gamma \vdash_T B\end{array}}{\Gamma \vdash_T A \wedge B}\wedge\mathrm{I}}{\Gamma \vdash_T A}\wedge\mathrm{E}_1$$

its reduced form p'

$$p' \quad = \quad \begin{array}{c}\vdots\, s \\ \Gamma \vdash_T A\end{array}$$

and the translation $[\![p]\!]_L$

$$[\![p]\!]_L \quad = \quad \dfrac{\dfrac{\begin{array}{cc}\vdots\, [\![s]\!]_L & \vdots\, [\![t]\!]_L \\ [\![\Gamma]\!]_L \vdash_R [\![A]\!]_L & [\![\Gamma]\!]_L \vdash_R [\![B]\!]_L\end{array}}{[\![\Gamma]\!]_L \vdash_R [\![A]\!]_L \,\&\, [\![B]\!]_L}\&\mathrm{I}}{[\![\Gamma]\!]_L \vdash_R [\![A]\!]_L}\&\mathrm{E}_1$$

It is easily verifiable that the commutativity holds

$$\begin{aligned}[\![p]\!]'_L \quad &= \quad \begin{array}{c}\vdots\, [\![s]\!]_L \\ [\![\Gamma]\!]_L \vdash_R [\![A]\!]_L\end{array} \\ &= \quad [\![p']\!]_L\end{aligned}$$

The proof for implication reduction involves both lollipop and exponential reductions. First, take a general reducible proof p

$$p \quad = \quad \dfrac{\dfrac{\dfrac{\begin{array}{c}\left(\dfrac{}{A \vdash_T A}\mathrm{Id}\right)\cdots \\ \vdots\, s \\ \Gamma, A \cdots \vdash_T B\end{array}}{\Gamma, A \vdash_T B}}{\Gamma \vdash_T A \to B}\to\mathrm{I} \quad \begin{array}{c}\vdots\, t \\ \Delta \vdash_T A\end{array}}{\Gamma, \Delta \vdash_T B}\to\mathrm{E}$$

its reduced form p'

$$p' \quad = \quad \dfrac{\begin{array}{c}\left(\begin{array}{c}\vdots\, t \\ \Delta \vdash_T A\end{array}\right)\cdots \\ \vdots\, s \\ \Gamma, \Delta \cdots \vdash_T B\end{array}}{\Gamma, \Delta \vdash_T B}$$

and the translation $[\![p]\!]_L$

$$
[\![p]\!]_L \quad = \quad
\dfrac{\dfrac{\dfrac{\Big(\dfrac{}{[[\![A]\!]_L] \vdash_R [\![A]\!]_L}\,[\mathrm{Id}]\Big)\cdots}{\vdots\ [\![s]\!]_L}}{\dfrac{[\![\Gamma]\!]_L, [[\![A]\!]_L] \cdots \vdash_R [\![B]\!]_L}{[\![\Gamma]\!]_L, [[\![A]\!]_L] \vdash_R [\![B]\!]_L}}
\qquad
\dfrac{\dfrac{}{\langle ![\![A]\!]_L\rangle \vdash_R ![\![A]\!]_L}\,\langle\mathrm{Id}\rangle}{}\,!\mathrm{E}
\qquad
\dfrac{\vdots\ [\![t]\!]_L}{}
}{
\dfrac{
\dfrac{[\![\Gamma]\!]_L, \langle ![\![A]\!]_L\rangle \vdash_R [\![B]\!]_L}{[\![\Gamma]\!]_L \vdash_R ![\![A]\!]_L \multimap [\![B]\!]_L}\multimap \mathrm{I}
\qquad
\dfrac{[\![\Delta]\!]_L \vdash_R [\![A]\!]_L}{[\![\Delta]\!]_L \vdash_R ![\![A]\!]_L}\,!\mathrm{I}
}{
[\![\Gamma]\!]_L, [\![\Delta]\!]_L \vdash_R [\![B]\!]_L
}\multimap \mathrm{E}
}
$$

We can define the translation of the implication reduction as first reducing the lollipop, and subsequently reducing the exponential, as in the sequence

$$
\begin{aligned}
[\![p]\!]_L \quad &\underset{\multimap}{\Rightarrow} \quad
\dfrac{\dfrac{\dfrac{\Big(\dfrac{}{[[\![A]\!]_L] \vdash_R [\![A]\!]_L}\,[\mathrm{Id}]\Big)\cdots}{\vdots\ [\![s]\!]_L}}{\dfrac{[\![\Gamma]\!]_L, [[\![A]\!]_L] \cdots \vdash_R [\![B]\!]_L}{[\![\Gamma]\!]_L, [[\![A]\!]_L] \vdash_R [\![B]\!]_L}}
\qquad
\dfrac{\vdots\ [\![t]\!]_L}{[\![\Delta]\!]_L \vdash_R [\![A]\!]_L}\,!\mathrm{I}
}{
[\![\Gamma]\!]_L, [\![\Delta]\!]_L \vdash_R [\![B]\!]_L
}\,!\mathrm{E}
\\
&\underset{!}{\Rightarrow} \quad
\dfrac{\dfrac{\begin{pmatrix}\vdots\ [\![t]\!]_L\\ [\![\Delta]\!]_L \vdash_R [\![A]\!]_L\end{pmatrix}\cdots}{\vdots\ [\![s]\!]_L}}{\dfrac{[\![\Gamma]\!]_L, [\![\Delta]\!]_L \cdots \vdash_R [\![B]\!]_L}{[\![\Gamma]\!]_L, [\![\Delta]\!]_L \vdash_R [\![B]\!]_L}}
\\
&= \quad [\![p]\!]'_L
\end{aligned}
$$

Writing down the translation of p', we can see that the two conclusions are equal.

$$
\begin{aligned}
[\![p']\!]_L \quad &= \quad
\dfrac{\dfrac{\begin{pmatrix}\vdots\ [\![t]\!]_L\\ [\![\Delta]\!]_L \vdash_R [\![A]\!]_L\end{pmatrix}\cdots}{\vdots\ [\![s]\!]_L}}{\dfrac{[\![\Gamma]\!]_L, [\![\Delta]\!]_L \cdots \vdash_R [\![B]\!]_L}{[\![\Gamma]\!]_L, [\![\Delta]\!]_L \vdash_R [\![B]\!]_L}}
\\
&= \quad [\![p]\!]'_L
\end{aligned}
$$

Because all proof reductions are composed of sequenced implication and conjunction reductions, it follows that the defined translation commutes with every proof reduction.

Chapter 2

Type theory

Type theory is the study of types, and it serves as a constructive way of organizing mathematical objects. Types are descriptions of constructions, and in a constructive system, every existing object needs a recipe for how it can be constructed. It follows that every mathematical object has an associated type.

To assert that a mathematical object a is of a certain type T, we write $a : T$, and this statement is called a **typing judgment**, or sometimes simply a **typing**. Analogously to judgments in logic, a typing judgment might be valid only in a certain context, so we introduce a notion of **contextualized typing judgments**, which have the form $\Gamma \vdash a : T$, meaning that a is of type T in the context Γ.

Traditionally, defining a type is a procedure consisting of fours steps (Bauer, 2019). First, the **formation** rules are given, which describe the conditions for a mathematical object T to be called a type. Then, the **introduction** rules specify how objects of this type are constructed. After an object is constructed, the **elimination** rules give ways of taking it apart. Lastly, objects that have type T may relate to each other in some ways, and these relationships are described by additional **equations**.

A collection of types is called a **type system**. One such type system is the *simply typed λ-calculus*, or STLC for short. It uses syntax of the untyped λ-calculus, and a metalanguage similar to natural deduction to describe its types. The version of STLC used in this thesis is the traditional simply typed λ-calculus, extended with product types.

Given a collection of base types, an STLC system is generated by introducing function and product types.

The formation rules of function types and product types are almost identical, so we present them both at the same time.

If A and B are types, then $(A \to B)$ is a type, and $(A \times B)$ is a type.

Types in STLC are then described by the grammar

$$A, B ::= X \mid (A \to B) \mid (A \times B)$$

for X ranging over base types.

Objects of STLC are **well-typed** terms of the untyped λ-calculus. A well-typed term is a term that is obtainable by deductions of the type system. A well-typed term is also called

$$\frac{\Gamma, x : A, y : B, \Delta \vdash s : C}{\Gamma, y : B, x : A, \Delta \vdash s : C}\,\text{Exchange}$$

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

$$\frac{\Gamma, x : A, y : A \vdash s : B}{\Gamma, z : A \vdash s[x := z][y := z] : B}\,\text{Contraction}$$

$$\frac{}{x : A \vdash x : A}\,\text{Id}$$

Figure 2.1: Structural rules and the identity axiom for STLC

a **program**. A context of a contextualized typing judgment in STLC is a list of typing judgments, where the terms being typed are variables, and every variable appears in the context at most once. When concatenating contexts, it is implicitly assumed that they do not share any variables.

Analogues to the structural rules from intuitionistic logic exist for STLC. The only difference is that the type-theoretical variants provide additional information on their action on terms. All three rules are listed in Figure 2.1, along with the identity axiom.

The exchange rule remains mostly unchanged. It asserts that changing the order of variable typings in the context has no effect on either the typed term or its type.

The weakening rule plays the same role as logical weakening, but it also states that the conclusion deduces the same term of the same type as the premise. As stated above, there is an implicit assumption that the variable x is not contained in the context Γ.

The contraction rule expresses that the type of an expression does not depend on specific values of its free variables, only their types. That is to say, any two variables x and y of the same type may be replaced by a new variable z without changing the resulting type. It employs capture-avoiding variable substitution as defined in (Sørensen & Urzyczyn, 2006), which is a metaoperation — the symbols '[', ':=' and ']' are not part of the language of lambda calculus. The metaterm $s[x := z]$ stands for the term s with free occurrences of the variable x replaced by the term z.

The identity axiom claims that every variable from the context can be derived.

The introduction and elimination rules for function types mirror the structure of implication deduction rules in intuitionistic logic. Whereas the logical interpretation relied on hypotheses, the type-theoretical interpretation is given in terms of binding variables and applying abstractions. The premise of the introduction rule presents a term s, and among its free variables might be the variable x (x is free in s if it was derived using the identity axiom, or it might not be referenced in s if it was derived using weakening). The conclusion then produces a λ-term which explicitly binds this variable.

The elimination rule introduces an application term, and together with β-reduction it gives a notion of "computation", which corresponds to the implication proof reduction rule.

$$\frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash (\lambda x.s) : (A \to B)} \to \mathrm{I} \qquad \frac{\Gamma \vdash f : (A \to B) \quad \Delta \vdash s : A}{\Gamma, \Delta \vdash (f\ s) : B} \to \mathrm{E}$$

β-reduction: $((\lambda x.s)\ t) \equiv s[x := t]$
η-conversion: $(\lambda x.(f\ x)) \equiv f$ when x is not free in f

Figure 2.2: Rules and equations of the function type

$$\frac{\Gamma \vdash s : A \quad \Gamma \vdash t : B}{\Gamma \vdash (s, t) : (A \times B)} \times \mathrm{I} \qquad \frac{\Gamma \vdash s : (A \times B)}{\Gamma \vdash (\mathsf{fst}\, s) : A} \times \mathrm{E}_1 \qquad \frac{\Gamma \vdash s : (A \times B)}{\Gamma \vdash (\mathsf{snd}\, s) : B} \times \mathrm{E}_2$$

β-reduction: $(\mathsf{fst}\,(s, t)) \equiv s$
β-reduction: $(\mathsf{snd}\,(s, t)) \equiv t$
η-conversion: $((\mathsf{fst}\, s)\,, (\mathsf{snd}\, s)) \equiv s$

Figure 2.3: Rules and equations of the product type

As a consequence of the η-conversion, we know that every object of the function type is equivalent to a λ-term. The rules and equations are listed in Figure 2.2.

The condition of x not being free in f for the η-conversion can be justified by looking at the expanded form of the equality, which is obtained by annotating the terms with their proof trees.

$$\begin{array}{c} \vdots \\ \Gamma \vdash f : (A \to B) \end{array} \quad \equiv \quad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash f : (A \to B)\end{array} \quad \dfrac{}{x : A \vdash x : A}\,\mathrm{Id}}{\Gamma, x : A \vdash (f\ x) : B} \to \mathrm{E}}{\Gamma \vdash (\lambda x.(f\ x)) : (A \to B)} \to \mathrm{I}$$

We see that the tree contains a typing in the context $\Gamma, x : A$. If x was free in f, then the list Γ would already contain a typing of the variable x, leading to a proof that is not well-formed.

On the other hand, the introduction and elimination rules for product types looks exactly like the ones for logical conjunction. Previously, we saw that conjunction in intuitionistic logic encodes the availability of proofs of both of its constituents, and this notion is made explicit as the product involves storing both terms. The elimination rules with β-reduction say that either of the two original terms may be recovered, and the reductions correspond to proof reduction of intuitionistic conjunction. The η-conversion for product types fulfills the same role as the one for function types — we see that every object of a product type is equivalent to a term constructed with the $\times$I rule. The rules and equations are listed in Figure 2.3.

The syntax of the terms of STLC is generated by the following grammar:

$$\begin{aligned} s, t ::= {} & x \\ | \quad & (\lambda x.s) \mid (s\ t) \\ | \quad & (s, t) \mid (\mathsf{fst}\ s) \mid (\mathsf{snd}\ s) \end{aligned}$$

for x ranging over variables.

The resemblance between STLC and intuitionistic logic is striking, and it has a name: the *Curry-Howard correspondence*. We can see a correspondence on three different levels.

First, the propositions from intuitionistic logic correspond to types. The judgment A *true* amounts to having an appropriate term s for which we can make the typing judgment $s : A$.

Second, every logical rule in intuitionistic logic has an equivalent in STLC, and every rule has an associated syntactic construct. Consequently, the term encodes the deduction tree that led to its construction, up to commuting structural rules. In other words, programs are proofs.

Lastly, the β-reduction rules, which are computational in nature, correspond to proof reductions. Therefore, computation is proof reduction.

2.1 Linear types

Given the correspondence between intuitionistic logic and the simply typed λ-calculus, we might wonder if there is a programming language corresponding to linear logic, and indeed there is.

In this section, we introduce a programming language called *linear λ-calculus*, or LLC. Its form is given by assigning terms to the logical deduction rules of linear logic. The syntax was influenced by (Wadler, 1993) and (Barber, 1996).

The context of contextualized judgments in LLC is a list of type judgments, each of which is enclosed in either square brackets $[\_]$, indicating an intuitionistic assumption, or angle brackets $\langle\_\rangle$, indicating a linear assumption. As with STLC, the terms typed in assumptions must be variables, and each variable can appear in the context at most once.

The type system includes two axioms, one for every kind of assumption, and they are used for introducing variables. The structural rules are almost identical to the ones of STLC, with the exception that they only act on intuitionistic assumptions. The axioms and structural rules are listed in Figure 2.4. The Exchange rule does not show any brackets around its assumptions, which is done to indicate that any two assumptions can be exchanged. This syntactic deviation is made in the name of not having to specify four separate exchange rules, one for each combination of an intuitionistic/linear pair.

We can see how the contraction rule allows an intuitionistic variable to be used more than once in a proof — instances of two separate intuitionistic assumptions of the same type can be replaced by one variable.

The rules of linear functions assign terms to $\multimap$ introduction and elimination, producing linear abstraction and linear application. As a consequence of λ-terms being formed strictly

$$\frac{}{\langle x : A\rangle \vdash x : A}\,\langle\text{Id}\rangle \qquad \frac{}{[x : A] \vdash x : A}\,[\text{Id}]$$

$$\frac{\Gamma, x : A, y : B, \Delta \vdash s : C}{\Gamma, y : B, x : A, \Delta \vdash s : C}\,\text{Exchange} \qquad \frac{\Gamma \vdash s : B}{\Gamma, [x : A] \vdash s : B}\,\text{Weakening}$$

$$\frac{\Gamma, [x : A], [y : A] \vdash s : B}{\Gamma, [z : A] \vdash s[x := z][y := z] : B}\,\text{Contraction}$$

Figure 2.4: Structural rules and identity axioms for LLC

$$\frac{\Gamma, \langle x : A\rangle \vdash s : B}{\Gamma \vdash (\lambda x.s) : (A \multimap B)} \multimap \text{I} \qquad \frac{\Gamma \vdash f : (A \multimap B) \quad \Delta \vdash s : A}{\Gamma, \Delta \vdash (f\ s) : B} \multimap \text{E}$$

β-reduction: $((\lambda x.s)\ t) \equiv s[x := t]$
η-conversion: $(\lambda x.(f\ x)) \equiv f$ when x is not free in f

Figure 2.5: Rules and equations for the $\multimap$ function type

by binding linear variables, we can conclude that every variable bound by a λ-term is used exactly once in its body. Therefore, the β-reduction is correct, meaning that the term t being substituted will end up being used exactly once. Both rules and equations for linear functions are listed in Figure 2.5.

The & product's terms and equations correspond to the $\times$ product from STLC. The introduction rule is used for forming a tuple of two resources, each of which references the exact same context, and the elimination rules allow the consumer to pick which component they want. The reduction equations identify components of a tuple with the objects extracted using the eliminators, and the conversion equation identifies every object of a & type with one constructed using the introduction rule. The rules and equations are listed in Figure 2.6.

The & product is the reason why we differentiate between "using" a variable exactly once in a program, and having the variable "appear" exactly once in a program. In the contextualized typing judgment

$$x : A \vdash (x, x) : (A \,\&\, A)$$

the variable x is *used* exactly once, because a & product can only be used by extracting one of its components, even though x *appears* twice in the program.

A tuple of two independent resources s and t is an instance of a $\otimes$ product, and it is written $|x, y|$. This syntax was chosen to indicate that the two resources exist "in parallel", in contrast to the & product. The elimination rule specifies a syntactic construction known in functional languages as *pattern matching* — if the term s assumes two linear variables x and y, then a value of the appropriate $\otimes$ type can be deconstructed into its parts, and the construct binds each part to the corresponding variable. Since they are both linear variables,

$$\frac{\Gamma \vdash s : A \quad \Gamma \vdash t : B}{\Gamma \vdash (s,t) : (A \,\&\, B)}\,\&\mathrm{I} \qquad \frac{\Gamma \vdash s : (A \,\&\, B)}{\Gamma \vdash (\mathsf{fst}\, s) : A}\,\&\mathrm{E}_1 \qquad \frac{\Gamma \vdash s : (A \,\&\, B)}{\Gamma \vdash (\mathsf{snd}\, s) : B}\,\&\mathrm{E}_2$$

$$\begin{aligned}&\beta\text{-reduction:}\ \ (\mathsf{fst}\,(s,t)) \equiv s\\ &\beta\text{-reduction:}\ \ (\mathsf{snd}\,(s,t)) \equiv t\\ &\eta\text{-conversion:}\ \ ((\mathsf{fst}\, s)\,,(\mathsf{snd}\, s)) \equiv s\end{aligned}$$

Figure 2.6: Rules and equations for the & product type

$$\frac{\Gamma \vdash s : A \quad \Delta \vdash t : B}{\Gamma, \Delta \vdash |s,t| : (A \otimes B)}\otimes\mathrm{I} \qquad \frac{\Gamma, \langle x : A\rangle, \langle y : B\rangle \vdash s : C \quad \Delta \vdash t : (A \otimes B)}{\Gamma, \Delta \vdash (\mathsf{case}\, t\, \mathsf{of}\, |x,y|\, \mathsf{in}\, s) : C}\otimes\mathrm{E}$$

$$\begin{aligned}&\beta\text{-reduction:}\ \ (\mathsf{case}\, |s,t|\, \mathsf{of}\, |x,y|\, \mathsf{in}\, u) \equiv u[x := s][y := t]\\ &\eta\text{-conversion:}\ \ (\mathsf{case}\, s\, \mathsf{of}\, |x,y|\, \mathsf{in}\, |x,y|) \equiv s\end{aligned}$$

Figure 2.7: Rules and equations for the $\otimes$ product type

the β-reduction once again preserves the property of using the components of a $\otimes$ product exactly once. The rules and equations are listed in Figure 2.7.

The η-conversion for the $\otimes$ product differs from what we have seen so far — past conversion were, in some sense, direct opposites of the corresponding β-reductions. Where β-reductions allowed to remove an introduction followed by an elimination, the η-conversions allowed wrapping a proof into an elimination followed by an introduction. On the other hand, annotating the terms of the η-conversion for $\otimes$ products gives the following diagram.

$$\begin{array}{c}\vdots\\ \Gamma \vdash s : (A \otimes B)\end{array} \quad\equiv\quad \dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash s : (A \otimes B)\end{array} \qquad \dfrac{\dfrac{}{\langle x : A\rangle \vdash x : A}\langle\mathrm{Id}\rangle \quad \dfrac{}{\langle y : B\rangle \vdash y : B}\langle\mathrm{Id}\rangle}{\langle x : A\rangle, \langle y : B\rangle \vdash |x,y| : (A \otimes B)}\otimes\mathrm{I}}{\Gamma \vdash (\mathsf{case}\, s\, \mathsf{of}\, |x,y|\, \mathsf{in}\, |x,y|) : (A \otimes B)}\otimes\mathrm{E}$$

We can see that the order of introduction/elimination is reversed for this conversion. Note, however, that this proof tree is not subject to β-reduction, because the introduction and elimination rules act on different instances of the connective — we introduce the term $|x,y|$, but eliminate the term s. The reason for this pattern change is that the $\otimes$ product has a different *polarity*. While all the previous types were *negative types*, the $\otimes$ product is *positive*. Intuitively, positive types encode structure, while negative types encode behavior. Exploration of type polarity is, however, out of scope for this thesis, therefore the interested reader may consult (Zeilberger, 2009).

The ! exponential is another example of a positive type. The pattern matching term $(\mathsf{case}\, t\, \mathsf{of}\, !x\, \mathsf{in}\, s)$ decomposes a source t into its "template" x, which can then be used however many times is necessary in the program s. The rules and equations are listed in Figure 2.8.

At first sight, it might not be obvious why the β-reduction holds. After all, the substitution might result in t being used non-linearly. Upon further inspection, we see that since

$$\frac{[\Gamma] \vdash s : A}{[\Gamma] \vdash !s : !A}\,!\mathrm{I} \qquad \frac{\Gamma, [x : A] \vdash s : B \quad \Delta \vdash t : !A}{\Gamma, \Delta \vdash (\mathsf{case}\, t\, \mathsf{of}\, !x\, \mathsf{in}\, s)}\,!\mathrm{E}$$

$$\beta\text{-reduction:}\ \ (\mathsf{case}\, !t\, \mathsf{of}\, !x\, \mathsf{in}\, s) \equiv s[x := t]$$
$$\eta\text{-conversion:}\ \ (\mathsf{case}\, s\, \mathsf{of}\, !x\, \mathsf{in}\, !x) \equiv s$$

Figure 2.8: Rules and equations for the ! exponential type

t was promoted to $!t$ using the !I rule, it can only use intuitionistic variables. Therefore, the substitution cannot break any linearity contracts.

$$
\begin{aligned}
x[x := s] &= s \\
y[x := s] &= y \\
(\lambda x.t)[x := s] &= (\lambda x.t) \\
(\lambda y.t)[x := s] &= (\lambda y.t[x := s]) \\
(t\ u)[x := s] &= (t[x := s]\ u[x := s]) \\
(t, u)\,[x := s] &= (t[x := s], u[x := s]) \\
(\mathsf{fst}\, t)\,[x := s] &= (\mathsf{fst}\, t[x := s]) \\
(\mathsf{snd}\, t)\,[x := s] &= (\mathsf{snd}\, t[x := s]) \\
|t, u|\,[x := s] &= |t[x := s], u[x := s]| \\
(\mathsf{case}\, t\, \mathsf{of}\, |x, y|\, \mathsf{in}\, u)\,[x := s] &= (\mathsf{case}\, t[x := s]\, \mathsf{of}\, |x, y|\, \mathsf{in}\, u) \\
(\mathsf{case}\, t\, \mathsf{of}\, |y, x|\, \mathsf{in}\, u)\,[x := s] &= (\mathsf{case}\, t[x := s]\, \mathsf{of}\, |y, x|\, \mathsf{in}\, u) \\
(\mathsf{case}\, t\, \mathsf{of}\, |y, z|\, \mathsf{in}\, u)\,[x := s] &= (\mathsf{case}\, t[x := s]\, \mathsf{of}\, |y, z|\, \mathsf{in}\, u[x := s]) \\
(!t)[x := s] &= !(t[x := s]) \\
(\mathsf{case}\, t\, \mathsf{of}\, !x\, \mathsf{in}\, u)\,[x := s] &= (\mathsf{case}\, t[x := s]\, \mathsf{of}\, !x\, \mathsf{in}\, u) \\
(\mathsf{case}\, t\, \mathsf{of}\, !y\, \mathsf{in}\, u)\,[x := s] &= (\mathsf{case}\, t[x := s]\, \mathsf{of}\, !y\, \mathsf{in}\, u[x := s])
\end{aligned}
$$

Figure 2.9: Substitution of LLC terms

The syntax of LLC is generated by the following grammar:

$$
\begin{aligned}
s, t ::= {} & x \\
\mid {} & (\lambda x.s) \mid (s\ t) \\
\mid {} & (s, t) \mid (\mathsf{fst}\, s) \mid (\mathsf{snd}\, s) \\
\mid {} & |s, t| \mid (\mathsf{case}\, s\, \mathsf{of}\, |x, y|\, \mathsf{in}\, t) \\
\mid {} & !s \mid (\mathsf{case}\, s\, \mathsf{of}\, !x\, \mathsf{in}\, t)
\end{aligned}
$$

for x and y ranging over variables.

Substitution of LLC terms is defined analogously to substitution of STLC terms. The rules are listed in Figure 2.9, and they are defined for distinct variables x, y and z, and for LLC terms s, t and u. The substitution avoids variable capture — if a free variable in s would become bound following the substitution, then the substitution is not defined, and renaming variables must precede.

2.2 Commuting conversions

In addition to equations describing relationships between introductions and eliminations of the same type, there are also equations relating rules of different types. These equations, called **commuting conversions**, describe valid ways of moving pattern matching terms through the program. For example, when given the program

$$(\mathsf{case}\, s\, \mathsf{of}\, |x, y|\, \mathsf{in}(f\ |x, y|))$$

one might feel that it ought to be equivalent to the program

$$(f\ s)$$

because the object s is being deconstructed only to be reconstructed in the same manner later, exactly like in the η-conversion rule for the $\otimes$ product. Alas, η-conversion is not applicable in this case, because the $\otimes$ elimination-introduction pair is interleaved with function application. Commuting conversions give us a framework for "tunneling" pattern matching, so that the first program can be rewritten to

$$(f\ \ (\mathsf{case}\, s\, \mathsf{of}\, |x, y|\, \mathsf{in}\, |x, y|))$$

where η-conversion is applicable.

We describe commuting conversions using terms-with-holes à la Barber[1] (Barber, 1996). A term-with-holes is a mathematical object defined by the grammar

$$\begin{aligned} C[\_], D[\_] ::= {} & \_ \\ | \quad & (\lambda x.C[\_]) \quad | (C[\_]\ s) \quad | (s\ C[\_]) \\ | \quad & (C[\_], D[\_]) \mid (\mathsf{fst}\, C[\_]) \mid (\mathsf{snd}\, C[\_]) \\ | \quad & |C[\_], s| \quad | |s, C[\_]| \quad | (\mathsf{case}\, C[\_]\, \mathsf{of}\, |x, y|\, \mathsf{in}\, s) \mid (\mathsf{case}\, s\, \mathsf{of}\, |x, y|\, \mathsf{in}\, C[\_]) \\ | \quad & (\mathsf{case}\, C[\_]\, \mathsf{of}\, !x\, \mathsf{in}\, s) \quad | (\mathsf{case}\, s\, \mathsf{of}\, !x\, \mathsf{in}\, C[\_]) \end{aligned}$$

for x and y ranging over variables and s ranging over terms of LLC.

In effect, a term-with-holes is a program constructed without the use of the !I rule, with a subterm replaced by '_', called a hole. Note that a term-with-holes $C[\_]$ uses exactly one hole. Emphasis is once again on the terminology "uses", because multiple holes may appear in a term-with-holes, if it was constructed using the $(C[\_], D[\_])$ rule. Next, we provide a way to fill the hole — $C[s]$ is defined as the term $C[\_]$ with the hole replaced by the term s.

The commuting conversions are listed in Figure 2.10. The motivating example mentioned above is an application of the first commuting conversion, with the term-with-holes $C[\_]$ being equal to $(f\ \_)$, and the term s being equal to $|x, y|$.

The requirements on the bindings of variables arise naturally when one writes down the proof trees for the terms on the two sides of the equations. When considering the first commuting conversion, the proof tree of the left side has the form

$$\dfrac{\begin{array}{c}\Theta, \langle x : A\rangle, \langle y : B\rangle \vdash t : E \\ \vdots \\ \Gamma, \Theta, \langle x : A\rangle, \langle y : B\rangle \vdash C[t] : D \qquad \Delta \vdash s : (A \otimes B)\end{array}}{\Gamma, \Theta, \Delta \vdash (\mathsf{case}\, s\, \mathsf{of}\, |x, y|\, \mathsf{in}\, C[t]) : D}\otimes\mathrm{E}$$

and the right side has the form

$$\dfrac{\dfrac{\Theta, \langle x : A\rangle, \langle y : B\rangle \vdash t : E \qquad \Delta \vdash s : (A \otimes B)}{\Theta, \Delta \vdash (\mathsf{case}\, s\, \mathsf{of}\, |x, y|\, \mathsf{in}\, t) : E}\otimes\mathrm{E}}{\begin{array}{c}\vdots \\ \Gamma, \Theta, \Delta \vdash C[(\mathsf{case}\, s\, \mathsf{of}\, |x, y|\, \mathsf{in}\, t)] : D\end{array}}$$

[1] Barber and other authors call this concept "contexts", "term contexts" or "contexts-with-holes", but we prefer terms-with-holes to avoid overloading the word "context"

$$(\mathsf{case}\ s\ \mathsf{of}\ |x, y|\ \mathsf{in}\ C[t]) \equiv C[(\mathsf{case}\ s\ \mathsf{of}\ |x, y|\ \mathsf{in}\ t)] \quad \text{when } x \text{ and } y \text{ are not free in } C[\_] \text{ and when } C[\_] \text{ does not bind } x \text{ or } y$$

$$(\mathsf{case}\ s\ \mathsf{of}\ !x\ \mathsf{in}\ C[t]) \equiv C[(\mathsf{case}\ s\ \mathsf{of}\ !x\ \mathsf{in}\ t)] \quad \text{when } x \text{ is not free in } C[\_] \text{ and when C[\_] does not bind } x$$

Figure 2.10: Commuting conversions

Since the first derivation contains the context $\Gamma, \Theta, \langle x : A\rangle, \langle y : B\rangle$, we know that neither x nor y may appear in Γ, which contains the free variables of $C[\_]$. The second condition specifies that the x and y that are free in t are the same x and y that are bound by the pattern matching in the final term. The only way to make them differ would be if the proof tree of $C[\_]$ first bound them for t, and then introduced them as new variables. For example, the term-with-hole

$$(((\lambda y.(\lambda x.\_))\ x)\ y)$$

is not eligible for the commuting conversion, because moving the pattern matching into the hole would swap the values of x and y.

Similar reasoning is used for obtaining the conditions of the second commuting conversion.

2.3 Rationale for kinded assumptions

Equipped with an explicit notation for terms, representing proofs, we can see why this system distinguishes between intuitionistic assumptions and exponential types as two representations of free resources. While intuitionistic assumptions can be only *variables*, objects of exponential types can be entire programs, containing linear variables. One might be tempted to simplify LLC by removing intuitionistic assumptions and replacing them with assumptions of exponential types, for example giving rise to the alternative rule for contraction:

$$\frac{\Gamma, x : !A, y : !A \vdash s : B}{\Gamma, z : !A \vdash s[x := z][y := z] : B}\ !\text{Contraction}$$

but we will see that this system breaks linearity when applying β-reductions.

The best way to approach this topic is with an example. Let the type W represent a proposition "I have a cup of water", and the type G represent "I have a liter of gas in the tank of my car". Consequently, the types $!W$ represents a water source, because it can provide an unlimited amount of cups of water, and the type $G \multimap !W$ represents a procedure for obtaining a water source using a moving car. Specifically, imagine the variable $c : G$ being a car with gas in the tank, and the function $f : A \multimap !W$ being the ability to drive to a neighboring city, bringing back a water fountain. Then the program $(f\ c) : !W$ represents the fountain obtained by going into the other city, consuming the gas in the process.

We can derive the following program, which says that water from one such fountain can

be distributed into two fountains.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x : !W\rangle \vdash x : !W}\langle\mathrm{Id}\rangle \quad \dfrac{}{\langle y : !W\rangle \vdash y : !W}\langle\mathrm{Id}\rangle}{\langle x : !W\rangle, \langle y : !W\rangle \vdash |x, y| : (!W \otimes !W)}\otimes\mathrm{I}}{\langle z : !W\rangle \vdash |z, z| : (!W \otimes !W)}\,!\mathrm{Contraction}}{\vdash (\lambda z.\, |z, z|) : (!W \multimap (!W \otimes !W))}\multimap \mathrm{I} \qquad \begin{array}{c}\vdots\\ \langle f : (G \multimap !W)\rangle, \langle c : G\rangle \vdash (f\ c) : !W\end{array}}{\langle f : (G \multimap !W)\rangle, \langle c : G\rangle \vdash (\lambda z.\, |z, z|)\ (f\ c) : (!W \otimes !W)}\multimap \mathrm{E}$$

This program is subject to β-reduction, because of the sequence of $\multimap$ introduction and elimination. However, reducing the program leads to the typing

$$\langle f : (G \multimap !W)\rangle, \langle c : G\rangle \vdash |(f\ c), (f\ c)| : (!W \otimes !W)$$

which clearly does not produce a well-typed term, because the linear variables f and c are used twice. This is akin to taking two trips in the car, but only having fuel for one trip. In other words, the system would not be locally sound.

The term assignment with intuitionistic assumptions solves this problem by not allowing multiple-use variables to be bound as arguments to lambdas. Instead, the argument must be always linear, and later consumed by exponential pattern matching. That is to say, the above program is written

$$\langle f : (G \multimap !W)\rangle, \langle c : G\rangle \vdash (\mathsf{case}(f\ c)\,\mathsf{of}\,!z\,\mathsf{in}\,|z, z|) : (!W \otimes !W)$$

where the linear variables are correctly used exactly once.

Flavors of linear logic and their term assignments without intuitionistic assumptions exist, for example (Benton, Bierman, de Paiva, & Hyland, 1993b). In general, these variants provide term assignments for the structural rules also. This thesis presents the variant with kinded assumptions, for its closer resemblance to the rules of STLC.

2.4 Intuitionistic embedding, revisited

Type theory extends the proof-theoretical point of view through programs encoding proofs. Since we are able to embed intuitionistic logic into linear logic, we want to also embed STLC into LLC, and this embedding needs to follow two conditions. First, it needs to agree with the Curry-Howard correspondence — that is to say, embedding of types must behave the same as embedding of propositions, embedding of programs must behave the same as embedding of proofs and embedding of computations must behave the same as embedding of proof reductions. Secondly, equivalent programs in STLC must translate to equivalent programs in LLC. The term correspondence is done following (Wadler, 1993), and we present its inner working in detail.

The first condition is satisfied rudimentarily — we define the translations of types, programs and computations by adding terms to the proof trees used in defining their logical counterparts. Then, agreement with the Curry-Howard correspondence is reached by definition.

For the intuitionistic correspondence between types and propositions, we obtain the following type embedding

$$\begin{aligned}\llbracket X \rrbracket_L &= X \\ \llbracket A \times B \rrbracket_L &= \llbracket A \rrbracket_L \,\&\, \llbracket B \rrbracket_L \\ \llbracket A \to B \rrbracket_L &= !\llbracket A \rrbracket_L \multimap \llbracket B \rrbracket_L\end{aligned}$$

For translating programs, we find the terms corresponding to translating derivation rules of intuitionistic logic.

$$\begin{array}{ll}\llbracket x \rrbracket_L = x & \text{Id} \\ \llbracket (\lambda x.s) \rrbracket_L = (\lambda y.\, (\mathsf{case}\, y\, \mathsf{of}\, !x\, \mathsf{in}\, \llbracket s \rrbracket_L)) \text{ for } y \text{ not free in } s & \to \text{I} \\ \llbracket (s\ t) \rrbracket_L = (\llbracket s \rrbracket_L\ \ ![\![t]\!]_L) & \to \text{E} \\ \llbracket (s,t) \rrbracket_L = (\llbracket s \rrbracket_L \,, \llbracket t \rrbracket_L) & \times \text{I} \\ \llbracket (\mathsf{fst}\, s) \rrbracket_L = (\mathsf{fst}\, \llbracket s \rrbracket_L) & \times \text{E}_1 \\ \llbracket (\mathsf{snd}\, s) \rrbracket_L = (\mathsf{snd}\, \llbracket s \rrbracket_L) & \times \text{E}_2\end{array}$$

for x ranging over variables and s and t ranging over well-typed terms.

Showing commutativity of equations with translation first requires commutativity of substitution with translation. We need to show that for every well-typed term s from STLC, the following equation holds

$$\llbracket t[x := s] \rrbracket_L \equiv \llbracket t \rrbracket_L [x := \llbracket s \rrbracket_L]$$

This is accomplished using structural induction over terms of STLC.

This property trivially holds for variables. For x and y two distinct variables, we have the following equalities

$$\begin{aligned}\llbracket x[x := s] \rrbracket_L = \llbracket s \rrbracket_L = x[x := \llbracket s \rrbracket_L] = \llbracket x \rrbracket_L [x := \llbracket s \rrbracket_L] \\ \llbracket y[x := s] \rrbracket_L = \llbracket y \rrbracket_L = y = y[x := \llbracket s \rrbracket_L] = \llbracket y \rrbracket_L [x := \llbracket s \rrbracket_L]\end{aligned}$$

For the induction step, we assume that the property holds for every subterm, and we produce the following equalities, for x and y two distinct variables and s and t well-typed

terms in STLC

$$
\begin{aligned}
[\![(\lambda x.t)[x := s]]\!]_L &= [\![(\lambda x.t)]\!]_L \\
&= (\lambda y.\, (\mathsf{case}\, y\, \mathsf{of}\, !x\, \mathsf{in}\, [\![t]\!]_L)) \\
&= (\lambda y.\, (\mathsf{case}\, y[x := [\![s]\!]_L]\, \mathsf{of}\, !x\, \mathsf{in}\, [\![t]\!]_L)) \\
&= (\lambda y.\, (\mathsf{case}\, y\, \mathsf{of}\, !x\, \mathsf{in}\, [\![t]\!]_L)\, [x := [\![s]\!]_L]) \\
&= (\lambda y.\, (\mathsf{case}\, y\, \mathsf{of}\, !x\, \mathsf{in}\, [\![t]\!]_L))[x := [\![s]\!]_L] \\
&= [\![(\lambda x.t)]\!]_L\, [x := [\![s]\!]_L] \\
[\![(\lambda y.t)[x := s]]\!]_L &= [\![(\lambda y.t[x := s])]\!]_L \\
&= (\lambda z.\, (\mathsf{case}\, z\, \mathsf{of}\, !y\, \mathsf{in}\, [\![t[x := s]]\!]_L)) \\
&= (\lambda z.\, (\mathsf{case}\, z\, \mathsf{of}\, !y\, \mathsf{in}\, [\![t]\!]_L\, [x := [\![s]\!]_L])) \\
&= (\lambda z.\, (\mathsf{case}\, z[x := [\![s]\!]_L]\, \mathsf{of}\, !y\, \mathsf{in}\, [\![t]\!]_L\, [x := [\![s]\!]_L])) \\
&= (\lambda z.\, (\mathsf{case}\, z\, \mathsf{of}\, !y\, \mathsf{in}\, [\![t]\!]_L)\, [x := [\![s]\!]_L]) \\
&= (\lambda z.\, (\mathsf{case}\, z\, \mathsf{of}\, !y\, \mathsf{in}\, [\![t]\!]_L))[x := [\![s]\!]_L] \\
&= [\![(\lambda y.t)]\!]_L\, [x := [\![s]\!]_L] \\
[\![(t\ u)[x := s]]\!]_L &= [\![(t[x := s]\ u[x := s])]\!]_L \\
&= ([\![t[x := s]]\!]_L\ ![\![u[x := s]]\!]_L) \\
&= ([\![t]\!]_L\, [x := [\![s]\!]_L]\ ![\![u]\!]_L\, [x := [\![s]\!]_L]) \\
&= ([\![t]\!]_L\ ![\![u]\!]_L)[x := [\![s]\!]_L] \\
&= [\![(t\ u)]\!]_L\, [x := [\![s]\!]_L] \\
[\![(t, u)\, [x := s]]\!]_L &= [\![(t[x := s], u[x := s])]\!]_L \\
&= ([\![t[x := s]]\!]_L\, , [\![u[x := s]]\!]_L) \\
&= ([\![t]\!]_L\, [x := [\![s]\!]_L], [\![u]\!]_L\, [x := [\![s]\!]_L]) \\
&= ([\![t]\!]_L\, , [\![u]\!]_L)\, [x := [\![s]\!]_L] \\
&= [\![(t, u)]\!]_L\, [x := [\![s]\!]_L] \\
[\![(\mathsf{fst}\, t)\, [x := s]]\!]_L &= [\![(\mathsf{fst}\, t[x := s])]\!]_L \\
&= (\mathsf{fst}\, [\![t[x := s]]\!]_L) \\
&= (\mathsf{fst}\, [\![t]\!]_L\, [x := [\![s]\!]_L]) \\
&= (\mathsf{fst}\, [\![t]\!]_L)\, [x := [\![s]\!]_L] \\
&= [\![(\mathsf{fst}\, t)]\!]_L\, [x := [\![s]\!]_L] \\
[\![(\mathsf{snd}\, t)\, [x := s]]\!]_L &= [\![(\mathsf{snd}\, t[x := s])]\!]_L \\
&= (\mathsf{snd}\, [\![t[x := s]]\!]_L) \\
&= (\mathsf{snd}\, [\![t]\!]_L\, [x := [\![s]\!]_L]) \\
&= (\mathsf{snd}\, [\![t]\!]_L)\, [x := [\![s]\!]_L] \\
&= [\![(\mathsf{snd}\, t)]\!]_L\, [x := [\![s]\!]_L]
\end{aligned}
$$

Next, we need to show that equivalent STLC terms are translated to equivalent LLC terms. This is accomplished by showing that this property holds for every β-reduction and every η-conversion equation. That is, for every equation $s \equiv s'$, we need to show that $[\![s]\!]_L \equiv [\![s]\!]_L$.

Proofs of commutativity for the β-reductions are obtained by the corresponding proofs of commutativity for proof reductions.

$$\begin{aligned}
[\![((\lambda x.s)\ t)]\!]_L &= ([\![(\lambda x.s)]\!]_L\ \ ![\![t]\!]_L) \\
&= ((\lambda y.\,(\mathsf{case}\ y\ \mathsf{of}\ !x\ \mathsf{in}\ [\![s]\!]_L))\ ![\![t]\!]_L) \\
&\equiv (\mathsf{case}\ ![\![t]\!]_L\ \mathsf{of}\ !x\ \mathsf{in}\ [\![s]\!]_L) \\
&\equiv [\![s]\!]_L\,[x := [\![t]\!]_L] \\
&= [\![s[x := t]]\!]_L \\
[\![(\mathsf{fst}\,(s,t))]\!]_L &= (\mathsf{fst}\,[\![(s,t)]\!]_L) \\
&= (\mathsf{fst}\,([\![s]\!]_L\,, [\![t]\!]_L)) \\
&\equiv [\![s]\!]_L \\
[\![(\mathsf{snd}\,(s,t))]\!]_L &= (\mathsf{snd}\,[\![(s,t)]\!]_L) \\
&= (\mathsf{snd}\,([\![s]\!]_L\,, [\![t]\!]_L)) \\
&\equiv [\![t]\!]_L
\end{aligned}$$

We have not specified what η-conversions correspond to in logic, so the verification is not as simple as following existing proofs, however it is still straight-forward.

$$\begin{aligned}
[\![(\lambda x.(f\ x))]\!]_L &= (\lambda y.\,(\mathsf{case}\ y\ \mathsf{of}\ !x\ \mathsf{in}\ [\![(f\ x)]\!]_L)) \\
&= (\lambda y.\,(\mathsf{case}\ y\ \mathsf{of}\ !x\ \mathsf{in}([\![f]\!]_L\ \ ![\![x]\!]_L))) \\
&\equiv (\lambda y.([\![f]\!]_L\ \ (\mathsf{case}\ y\ \mathsf{of}\ !x\ \mathsf{in}\ !x))) \\
&\equiv (\lambda y.([\![f]\!]_L\ \ y)) \\
&\equiv [\![f]\!]_L \\
[\![((\mathsf{fst}\ s)\,, (\mathsf{snd}\ s))]\!]_L &= ([\![(\mathsf{fst}\ s)]\!]_L\,, [\![(\mathsf{snd}\ s)]\!]_L) \\
&= ((\mathsf{fst}\ [\![s]\!]_L)\,, (\mathsf{snd}\ [\![s]\!]_L)) \\
&\equiv [\![s]\!]_L
\end{aligned}$$

Since we covered all equations from STLC, we can conclude that equivalent programs translate to equivalent programs.

Chapter 3

Category theory

Category theory is the study of categories, which allow for expressing many mathematical structures and properties via diagrams of arrows between abstract objects (Mac Lane, 1998). Categorical semantics, as opposed to the traditional set-theoretical interpretations, have the advantage of being defined with more general structures, not limiting oneself to sets specifically (Crole, 1993, pp.129–132).

This chapter introduces only those categorical concepts that are necessary for giving categorical semantics to the linear lambda calculus, and it only gives a bird-eye view. For a broader introduction to category theory, see (Mac Lane, 1998) or (Adámek, Herrlich, & Strecker, 1990). The ultimate objective is to represent types as objects in a category, programs as arrows from a context object to a target object, type constructors as functors and rules as natural transformations.

A **category** is composed of two collections, the **objects** and the **arrows**, two operations on any arrow, the **source** and the **target**, and additional structure, specifically the **arrow composition** operator and the existence of **identity arrows**, which are the left and right identity for the composition.

The source and target operations are written src and tar, respectively, and each associates an object to an arrow — an arrow a with $\mathsf{src}(a) = X$ and $\mathsf{tar}(a) = Y$ is written graphically as $a : X \to Y$ or $X \xrightarrow{a} Y$. The collection of arrows between objects X and Y is written $\mathsf{hom}(X, Y)$.

The composition operator asserts that for any two arrows with matching ends, for example $a : X \to Y$ and $b : Y \to Z$, there exists an arrow $b \circ a : X \to Z$. This composition is associative, that is, the following equality holds for any three arrows $a : X \to Y$, $b : Y \to Z$, $c : Z \to W$ for any four objects X, Y, Z and W

$$c \circ (b \circ a) = (c \circ b) \circ a$$

Every object X is equipped with the identity arrow $\mathsf{id}_X : X \to X$. The identity arrows serve as identities for composition, which is described by the equations

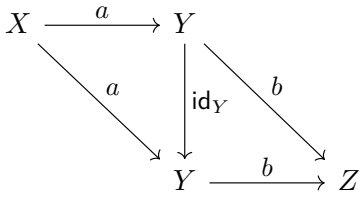
$$\mathsf{id}_{\mathsf{tar}(a)} \circ a = a$$
$$a \circ \mathsf{id}_{\mathsf{src}(a)} = a$$

In situations where the object is apparent from context, we choose to omit the subscript and simply write id.

Examples of categories include **Set**, the category with all sets for objects and functions for arrows, or **Grp**, the category of all groups and homomorphisms between them.

For every category $\mathcal{C}$, there exists the **opposite category** $\mathcal{C}^{\mathsf{op}}$. The opposite category has the exact same objects and identity arrows as the original category, but the arrows are reversed — the collection of arrows $\mathsf{hom}(X, Y)$ in $\mathcal{C}$ corresponds to the collection of arrows $\mathsf{hom}(Y, X)$ in $\mathcal{C}^{\mathsf{op}}$, and the composition $a \circ b$ in $\mathcal{C}^{\mathsf{op}}$ corresponds to the composition $b \circ a$ in $\mathcal{C}$.

An important tool of category theory are commutative diagrams. A commutative diagram is a diagram for which commutativity is either assumed or proven. Such a diagram contains nodes, which represent objects, and directed edges, which represent arrows. For example, we can construct a commutative diagram which represents the property of the identity arrow being the identity of arrow composition.



This diagram commutes when the two triangles commute, and we can see that the left triangle expresses the equality $\mathsf{id}_Y \circ a = a$, and the right one $b \circ \mathsf{id}_Y = b$.

An arrow $a : X \to Y$ is an **isomorphism** if there is another arrow $a^{-1} : Y \to X$ such that the equations $a^{-1} \circ a = \mathsf{id}_X$ and $a \circ a^{-1} = \mathsf{id}_Y$ hold. Isomorphisms are also called *invertible* arrows.

In the remainder of this thesis, we will only consider **locally small categories**, which have the property that for every two objects X and Y, the collection $\mathsf{hom}(X, Y)$ is a set. For example the category **Set** is locally small, because although the objects form a proper class, there is only a set of functions between any two given sets.

We introduce the notion of structure-preserving mappings between categories, called **functors**. Given two categories $\mathcal{C}$ and $\mathcal{D}$, a functor F from $\mathcal{C}$ to $\mathcal{D}$, denoted $F : \mathcal{C} \to \mathcal{D}$, needs to map contents of $\mathcal{C}$ to contents of $\mathcal{D}$ while preserving the categorical structure — the contents of a category are its objects and arrows, and the structure is described by the source and target assignments, arrow composition and identity arrows.

A functor $F : \mathcal{C} \to \mathcal{D}$ consists of a mapping from objects of $\mathcal{C}$ to objects of $\mathcal{D}$ and another mapping from arrows of $\mathcal{C}$ to arrows of $\mathcal{D}$, the two of which interact in such a way that for any arrow a in $\mathcal{C}$, its image Fa is an arrow in $\mathcal{D}$ whose source and target are the images of the source and target of a, graphically the image of an arrow $a : X \to Y$ is $Fa : FX \to FY$. Additionally, the identity arrows in $\mathcal{C}$ map to identity arrows in $\mathcal{D}$, so that $F\,\mathsf{id}_X = \mathsf{id}_{FX}$ for all objects X in $\mathcal{C}$, and the compositions of arrows in $\mathcal{C}$ maps to composition of arrows in $\mathcal{D}$, following the equation $F(b \circ a) = Fb \circ Fa$. These laws are represented by the commutative diagrams in Figure 3.1.

Because we are interested in functors with multiple parameters, we also need to describe the concept of a **product category**. Given two categories $\mathcal{C}$ and $\mathcal{D}$, their product is denoted

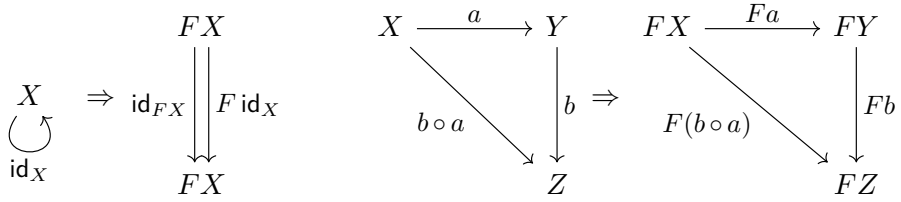


Figure 3.1: Functor laws

by $\mathcal{C} \times \mathcal{D}$. Objects of this category are ordered pairs (C, D), where C is an object of $\mathcal{C}$ and D is an object of $\mathcal{D}$, and arrows of this category are ordered pairs $(f, g) : (C, D) \to (C', D')$, where $f : C \to C'$ and $g : D \to D'$ are arrows of $\mathcal{C}$ and $\mathcal{D}$, respectively. Identity and composition are defined in the obvious way, the identity arrow being $\mathsf{id}_{(C,D)} = (\mathsf{id}_C, \mathsf{id}_D)$ and composition being $(f', g') \circ (f, g) = (f' \circ f, g' \circ g)$.

A functor $F : \mathcal{C} \times \mathcal{D} \to \mathcal{B}$ can be regarded as a mapping with two arguments, one from $\mathcal{C}$ and one from $\mathcal{D}$. For example, we can define the Cartesian product on sets as a functor $\_ \times \_ : \mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$. It takes a pair of two sets, (A, B), and maps it to the Cartesian product $A \times B$, which is itself a set. The action on arrows is defined element-wise. When the domain of a functor is not a binary product category, but a more general n-ary product category, then we talk about n-ary functors.

In the context of n-ary functors, we can refer to the property of being **functorial in an argument**. When saying that a functor $F : \mathcal{C} \times \mathcal{D} \to \mathcal{B}$ is *functorial in* $\mathcal{C}$, we mean that fixing any object D of $\mathcal{D}$ gives rise to a functor $\hat{F} : \mathcal{C} \to \mathcal{B}$, which is defined by its action on objects C of $\mathcal{C}$ by $\hat{F}C = F(C, D)$, and by its action on arrows $a : C \to C'$ by $\hat{F}a = F(a, \mathsf{id}_D) : F(C, D) \to F(C', D)$. This notion is extended from binary to n-ary functors, by implying that all arguments other than the one for which we check for functoriality are fixed. A functor is functorial in all its arguments, and conversely a mapping that is functorial in all its arguments is a functor.

An important functor we will use is the hom **functor**, $\mathsf{hom}(\_, \_) : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$, which to every pair of objects X and Y in $\mathcal{C}$ assigns their hom-set $\mathsf{hom}(X, Y)$. Functoriality in the second argument means that for every arrow $a : Y \to Y'$, we can find an arrow $\mathsf{hom}(\mathsf{id}_X, a) : \mathsf{hom}(X, Y) \to \mathsf{hom}(X, Y')$. Since the objects of **Set** are sets, and the arrows are functions on sets, we can show this action on elements of the hom-sets. An element of the set $\mathsf{hom}(X, Y)$ is an arrow $a : X \to Y$, and we need to transform it into an arrow $\mathsf{hom}(\mathsf{id}_X, f) : X \to Y$, which is easily done by post-composition with f, because for every arrow $a : X \to Y$, there is an arrow $f \circ a : X \to Y'$.

Functoriality in the first argument requires that it be from the opposite category, $\mathcal{C}^{\mathsf{op}}$. Only in this way can we define the action on arrows as pre-composition, analogously to functoriality in the second argument. We know that every arrow $f : X \to X'$ in $\mathcal{C}^{\mathsf{op}}$ is the arrow $f : X' \to X$ in $\mathcal{C}$. Then, finding the arrow $\mathsf{hom}(f, \mathsf{id}_Y) : \mathsf{hom}(X, Y) \to \mathsf{hom}(X', Y)$ is as easy as taking an element of the set $\mathsf{hom}(X, Y)$, which is an arrow $a : X \to Y$ in $\mathcal{C}$, and pre-composing it with the arrow $f : X' \to X$ in $\mathcal{C}$, to obtain the arrow $a \circ f : X' \to Y$, which is an element of the set $\mathsf{hom}(X', Y)$.

Furthermore, we will work with mappings between functors, called **natural transfor-**

mations. Given two categories $\mathcal{C}$ and $\mathcal{D}$, and two functors $F, G : \mathcal{C} \to \mathcal{D}$, a natural transformation $\varphi : F \Rightarrow G$ consists of a family of arrows $\varphi_C : FC \to GC$ (its **components**), indexed by objects of $\mathcal{C}$, such that the following diagram commutes for all objects X and Y, all arrows $f : X \to Y$ from $\mathcal{C}$.

$$\begin{array}{ccc} FX & \xrightarrow{Ff} & FY \\ \downarrow{\scriptstyle \varphi_X} & & \downarrow{\scriptstyle \varphi_Y} \\ GX & \xrightarrow{Gf} & GY \end{array}$$

When all the components of a natural transformation φ are isomorphisms, we call φ a **natural isomorphism**.

Similarly to functoriality in an argument, we can talk about **naturality in an argument** when n-ary functors are involved. Given two functors $F, G : \mathcal{C} \times \mathcal{D} \to \mathcal{B}$, a family of arrows $\varphi_{C,D} : F(C, D) \to G(C, D)$, indexed by objects of $\mathcal{C}$ and $\mathcal{D}$, is natural in C for a fixed object D in $\mathcal{D}$ if the family $\varphi_{C,D} : F(C, D) \to G(C, D)$, indexed by *only the objects* C of $\mathcal{C}$, is a natural transformation from $F(\_, D) : \mathcal{C} \to \mathcal{B}$ to $G(\_, D) : \mathcal{C} \to \mathcal{B}$, that is if the diagram

$$\begin{array}{ccc} F\,(C, D) & \xrightarrow{F\,(f, \mathsf{id}_D)} & F\,(C', D) \\ \downarrow{\scriptstyle \hat{\varphi}_C} & & \downarrow{\scriptstyle \hat{\varphi}_{C'}} \\ G\,(C, D) & \xrightarrow{G\,(f, \mathsf{id}_D)} & G\,(C', D) \end{array}$$

commutes for all arrows $f : C \to C'$ in $\mathcal{C}$.

Naturality of natural transformations between n-ary functors is defined in the obvious way, by asserting that fixing all arguments except the one of interest yields a natural transformation. A natural transformation is natural in all its arguments, and a mapping between functors that is natural in all arguments is a natural transformation.

The last tool in the basic categorical toolbox are **adjunctions**. While natural transformations represented a mapping between two parallel functors, an adjunction describes a correspondence between functors going in opposing directions. Given two categories $\mathcal{C}$ and $\mathcal{D}$, and functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$, we say that F is **left adjoint** to G (or equivalently that G is **right adjoint** to F) if there is a bijection between hom-sets $\psi_{X,Y} : \mathsf{hom}(FX, Y) \to \mathsf{hom}(X, GY)$ for all objects X in $\mathcal{C}$ and Y in $\mathcal{D}$, such that it is natural in X and Y. This relationship is written $F \dashv G$.

Naturality in X means that for any arrow $a : X' \to X$ in $\mathcal{C}$, the following diagram

$$\begin{array}{ccc} \mathsf{hom}(FX, Y) & \xrightarrow{\mathsf{hom}(Fa, Y)} & \mathsf{hom}(FX', Y) \\ \downarrow{\scriptstyle \psi_{X,Y}} & & \downarrow{\scriptstyle \psi_{X',Y}} \\ \mathsf{hom}(X, GY) & \xrightarrow{\mathsf{hom}(a, GY)} & \mathsf{hom}(X', GY) \end{array}$$

commutes, while naturality in Y means that for any arrow $a : Y \to Y'$ in $\mathcal{D}$, the diagram

$$\begin{array}{ccc} \mathsf{hom}(FX, Y) & \xrightarrow{\mathsf{hom}(FX,a)} & \mathsf{hom}(FX, Y') \\ \downarrow{\scriptstyle \psi_{X,Y}} & & \downarrow{\scriptstyle \psi_{X,Y'}} \\ \mathsf{hom}(X, GY) & \xrightarrow{\mathsf{hom}(X,Ga)} & \mathsf{hom}(X, GY') \end{array}$$

commutes.

There are other, equivalent, definitions of adjoint functors. We present one of them here, because the text to follow uses properties of the definition above as well as the one listed below. We leave out the standard proof that these two definitions are equivalent, referring to (Mac Lane, 1998).

An adjunction is also determined by a functor $F : \mathcal{C} \to \mathcal{D}$ and by the existence of "cofree" objects: for every object D in $\mathcal{D}$, there exists an object $G_0 D$ in $\mathcal{C}$ together with an arrow $\epsilon_D : FG_0 D \to D$, such that for every arrow $f : FC \to D$ in $\mathcal{D}$, there is a unique arrow $f^\flat : C \to G_0 D$, making the following diagram commute

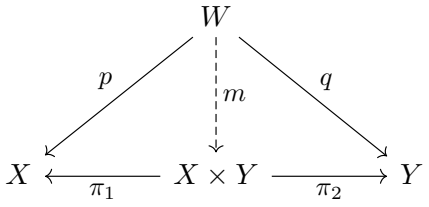
$$\begin{array}{ccc} D & \xleftarrow{\epsilon_D} & FG_0 D \\ & \nwarrow^{f} & \uparrow{\scriptstyle Ff^\flat} \\ & & FC \end{array}$$

3.1 Models of intuitionistic programs

For modeling the simply typed λ-calculus, we need to interpret types, contexts and programs of STLC, in a way that the interpretation preserves β and η equivalences.

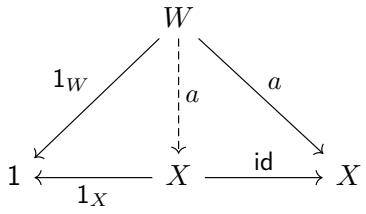
The types and contexts are represented by objects in a category, and programs are interpreted as arrows from context objects to target type objects — thus to interpret a contextualized type judgment $\Gamma \vdash t : A$, we first need an object $[\![\Gamma]\!]_C$, an object $[\![A]\!]_C$, and then an appropriate arrow $[\![t]\!]_C : [\![\Gamma]\!]_C \to [\![A]\!]_C$. Because the semantic brackets hinder readability of diagrams significantly, we choose to use the same name for a meta object in type theory and its categorical interpretation, so the above example will be henceforth written as $t : \Gamma \to A$ or $B \xrightarrow{t} A$.

The category for interpreting STLC needs to be equipped with additional structure. We begin by introducing the **product** in a category (which is different from a product of categories), a notion generalizing the Cartesian product from set theory. A product of two objects X and Y, usually denoted $X \times Y$, is another object in the same category equipped with two arrows, $\pi_1 : X \times Y \to X$ and $\pi_2 : X \times Y \to Y$, such that for all other objects W and arrows $p : W \to X$ and $q : W \to Y$, there is a unique arrow $m : W \to X \times Y$, called the **factoring arrow**, satisfying the equations $p = \pi_1 \circ m$ and $q : \pi_2 \circ m$. This property is captured in the commuting diagram



and the arrow m is also denoted $\langle p, q \rangle$.

We focus on categories that have all binary products, which means that the product $X \times Y$ exists for all objects X and Y of the category. We also require presence of the product unit — an object $\mathsf{1}$ in the same category, with the property that for every other object Z in the category, there exists a unique arrow $\mathsf{1}_Z : Z \to \mathsf{1}$. With this property alone, we can show that there is an isomorphism between the objects $X \times \mathsf{1}$ and X, since X can be shown to manifest the product property — for every other object W, there is only one arrow from it to $\mathsf{1}$, specifically $\mathsf{1}_W$, and then given an arrow $a : W \to X$, there is indeed only one arrow that is equal to $\mathsf{id} \circ a$, and that is a itself.



The symmetric isomorphism $\mathsf{1} \times X \cong X$ is provable in the same manner.

Categories with all binary products come equipped with a **product functor** $\_ \times \_ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, which to each pair of objects (C, D) associates their product $C \times D$, and acts on arrows in the obvious way.

This structure is enough for us to model contexts. Interpretation of the empty context is the unit $\mathsf{1}$, and context concatenation Γ, A is interpreted as the product $[\![\Gamma]\!]_C \times A$. Since the product is associative, we omit parentheses.

Composition of arrows reflects substitution — two arrows $s : A \to B$ and $t : B \to C$ represent two programs, the first program $x : A \vdash s : B$ possibly having a free variable of type A, and the second program $y : B \vdash t : C$ possibly having a free variable of type B. The "possibly" indicates that these variables might not appear in the program if they were introduced via weakening. Upon carefully choosing variable names to prevent accidental captures and conflicts, we interpret the substitution $x : A \vdash t[y := s] : C$ as the composition $t \circ s : A \to C$.

We proceed to give interpretations to the rules of STLC. The identity axiom requires that every object is equipped with an arrow going to itself, and this arrow is guaranteed to exist by the identity arrow property of all categories. Visually, we represent this rule as

$$\frac{}{A \xrightarrow{\mathsf{id}_A} A}\,\mathrm{Id}$$

The exchange rule is modeled by the symmetry of the product — there is an isomorphism $p_{X,Y} : X \times Y \to Y \times X$ for all objects X and Y, satisfying the property $p_{Y,X} \circ p_{X,Y} = \mathsf{id}_{X \times Y}$.

$$\frac{\Gamma \times A \times B \times \Delta \xrightarrow{s} B}{\Gamma \times B \times A \times \Delta \cong \Gamma \times A \times B \times \Delta \xrightarrow{s} B}\,\text{Exchange}$$

The contraction rule is given interpretation via the arrow $\langle \mathsf{id}_A, \mathsf{id}_A \rangle : A \to A \times A$.

$$\frac{\Gamma \times A \times A \xrightarrow{s} B}{\Gamma \times A \xrightarrow{\mathsf{id}_\Gamma \times \langle \mathsf{id}_A, \mathsf{id}_A \rangle} \Gamma \times A \times A \xrightarrow{s} B}\,\text{Contraction}$$

We can represent contraction, as well as all other deduction rules, via a natural transformation (Benton, Bierman, de Paiva, & Hyland, 1993a). We choose to present the premises and the conclusion of a rule with appropriate functors, whose arguments are the parts of the rule that "do not change" during the application of the rule, and which are parameterized by the objects that change. Specifically for the contraction rule, we choose the family of functors $P_A, C_A : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$ (for *premise functor* and *conclusion functor*), which are parameterized by the object A, and functorial in the objects Γ and B. The premise functors map the tuple (Γ, B) to the hom-set $\mathsf{hom}(\Gamma \times A \times A, B)$, and the conclusion functors map the tuple (Γ, B) to the hom-set $\mathsf{hom}(\Gamma \times A, B)$. The family of natural transformations φ^A have for components the arrows $\varphi^A_{\Gamma,B} : P_A(\Gamma, B) \to C_A(\Gamma, B)$.

This formulation gives rise to two naturality diagrams, one for the naturality in Γ and one for the naturality in B. In all natural transformations describing derivation rules, the naturality in context objects corresponds to the ability of replacing variables with expressions of the same type. We give a worked example for the contraction rule, and we do not describe the naturality in context objects for other rules.

Naturality in Γ is expressed by the following diagram

$$\begin{array}{ccc} \mathsf{hom}(\Gamma \times A \times A, B) & \xrightarrow{P_A(f, \mathsf{id}_B)} & \mathsf{hom}(\Gamma' \times A, B) \\ \downarrow{\scriptstyle \varphi^A_{\Gamma,B}} & & \downarrow{\scriptstyle \varphi^A_{\Gamma',B}} \\ \mathsf{hom}(\Gamma \times A, B) & \xrightarrow{C_A(f, \mathsf{id}_B)} & \mathsf{hom}(\Gamma' \times A, B) \end{array}$$

Because the components of φ^A are arrows in the category **Set**, we know that they correspond to functions between sets, so we can observe their action on elements of the sets. This action on elements is described in an *element chasing* diagram, shown in Figure 3.2.

The arrow in the upper left corner represents a program with possibly three free variables, one of the type Γ, and two of the type A. For explanation's sake we assign names to the variables, but keep in mind that the names can always be chosen to avoid accidental capture and conflicts. We say that the arrow s is an interpretation of a program $u : \Gamma, x : A, y : A \vdash s : B$. Following this program to the right, we pre-compose the arrow $f : \Gamma' \to \Gamma$, which represents a program $w : \Gamma' \vdash f : \Gamma$. Composition corresponds to substitution, so the

Figure 3.2: Element chasing for the naturality diagram of $\varphi^A_{\_,B}$

program represented by the arrow in the upper right is $w : \Gamma', x : A, y : A \vdash s[u := f] : B$. Chasing the programs vertically, we apply the contraction rule, so the bottom left arrow is the program $u : \Gamma, z : A \vdash s[x := z][y := z]$, and the bottom right arrow is the program $w : \Gamma', z : A \vdash (s[u := f])[x := z][y := z]$. However, the importance of the naturality property is that the bottom right program must be the same irrespective of the choice of the path. Transporting the bottom left arrow to the right, we obtain the program $w : \Gamma', z : A \vdash (s[x := z][y := z])[u := f]$, and the equation $(s[u := f])[x := z][y := z] = (s[x := z][y := z])[u := f]$. As mentioned, this equation corresponds to the fact variables in the context can be substituted.

Naturality in B gives rise to a similar equation, namely $(f[w := s])[x := z][y := z] = f[w := s[x := z][y := z]]$, which also holds by appropriately choosing variable names.

Weakening is defined with the projection arrow $\pi_1 : \Gamma \times B \to \Gamma$.

$$\frac{\Gamma \xrightarrow{s} A}{\Gamma \times B \xrightarrow{\pi_1} \Gamma \xrightarrow{s} A}\,\text{Weakening}$$

The premise and conclusion functors $P_B, C_B : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$ are connected via the natural transformations $\varphi^B : P_B \Rightarrow C_B$, and nothing of note is provided by the naturality.

The existence of all binary products plays the role of product type formation — if A and B are objects in $\mathcal{C}$, then $A \times B$ is also an object in $\mathcal{C}$. The product projections give meaning to the $\times$E rules, specifically

$$\frac{\Gamma \xrightarrow{s} A \times B}{\Gamma \xrightarrow{s} A \times B \xrightarrow{\pi_1} A}\times\mathrm{E}_1 \qquad \frac{\Gamma \xrightarrow{s} A \times B}{\Gamma \xrightarrow{s} A \times B \xrightarrow{\pi_2} A}\times\mathrm{E}_2$$

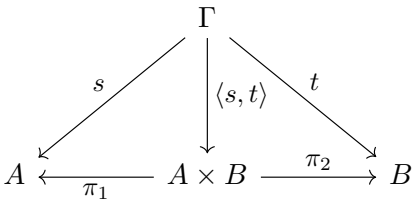
the family of functors indexed by A and B being $P_{A,B}, C_{A,B} : \mathcal{C}^{\mathsf{op}} \to \mathbf{Set}$ defined by $P_{A,B}(\Gamma) = \mathsf{hom}(\Gamma, A \times B)$ and $C_{A,B}(\Gamma) = \mathsf{hom}(\Gamma, A)$, and analogously for the other component. The functors only take one argument, the context.

The product introduction is formulated in terms of the factoring arrow, with the deduction

$$\frac{\Gamma \xrightarrow{s} A \qquad \Gamma \xrightarrow{t} B}{\Gamma \xrightarrow{\langle s,t\rangle} A \times B}\times\mathrm{I}$$

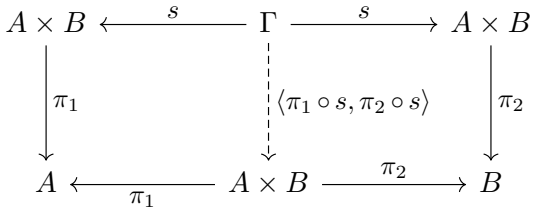
and functors $P_{A,B}, C_{A,B} : \mathcal{C}^{\mathsf{op}} \to \mathbf{Set}$ defined by $P_{A,B}(\Gamma) = \mathsf{hom}(\Gamma, A) \times \mathsf{hom}(\Gamma, B)$ and $C_{A,B}(\Gamma) = \mathsf{hom}(\Gamma, A \times B)$.

Equations are modeled with commutative diagrams. The β-reductions correspond exactly to the product property



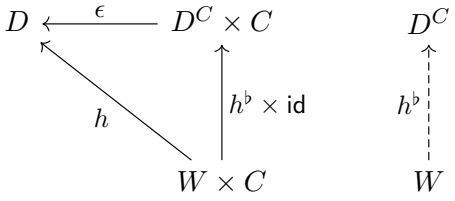
with the left triangle describing the equation $(\mathsf{fst}\,(s,t)) \equiv s$ and the right triangle describing the equation $(\mathsf{snd}\,(s,t)) \equiv t$.

The η-conversion corresponds to the diagram



Because of the product property, the dashed arrow is a unique arrow that makes the two squares commute. However, the arrow s also makes the squares commute, so the two arrows must be equal, making the terms they represent, s and $((\mathsf{fst}\,s)\,,(\mathsf{snd}\,s))$, equivalent.

To model function types, we introduce another categorical concept — the **exponential**. The exponential of two objects C and D in a category $\mathcal{C}$ is another object in the same category, denoted D^C, equipped with an arrow $\epsilon : D^C \times C \to D$ (called the **evaluator**), such that for all other objects W and arrows $h : W \times C \to D$, there is a unique arrow $h^\flat : W \to D^C$, satisfying the equation $\epsilon \circ (h^\flat \times \mathsf{id}_C) = h$. This property is captured in the commuting diagram



If a category has an exponential D^C for all objects C and D, then it gives rise to a functor $\_^- : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$, and the property corresponds to an adjunction with the product functor — for every object C in $\mathcal{C}$, the functor $\_ \times C$ is left adjoint to the functor $\_^C$. In other words, there is a bijection between the hom-sets $\mathsf{hom}(B \times C, D)$ and $\mathsf{hom}(B, D^C)$.

A category equipped with all binary products, a unit object for the product, and all exponentials, is called a **Cartesian closed category**. This is all the structure necessary for interpreting STLC, which we show by interpreting rules and equations for the function type via exponentials.

The $\to$ I rule takes advantage of the bijection of hom-sets

$$\frac{\Gamma \times A \xrightarrow{s} B}{\Gamma \xrightarrow{s^\flat} B^A} \to \mathrm{I}$$

with the natural transformation between $P_{A,B}(\Gamma) = \mathsf{hom}(\Gamma \times A, B)$ and $C_{A,B}(\Gamma) = \mathsf{hom}(\Gamma, B^A)$ being the natural isomorphism with hom-set bijections for components.

The $\to$ E rule uses the ϵ evaluator

$$\frac{\Gamma \xrightarrow{s} B^A \qquad \Delta \xrightarrow{t} A}{\Gamma \times \Delta \xrightarrow{s \times t} B^A \times A \xrightarrow{\epsilon} B} \to \mathrm{E}$$

with the obvious premise and conclusion functors taking for arguments Γ and Δ.

The β-reduction equation is expressed via the commutative diagram

$$\begin{array}{ccc} \Gamma \times \Delta & \xrightarrow{\mathsf{id} \times t} & \Gamma \times A \\ {\scriptstyle s^\flat \times t}\downarrow & & \downarrow{\scriptstyle s} \\ B^A \times A & \xrightarrow[\epsilon]{} & B \end{array}$$

where we decompose the arrow $\Gamma \times \Delta \xrightarrow{s^\flat \times t} B^A \times A$ as $\Gamma \times \Delta \xrightarrow{\mathsf{id} \times t} \Gamma \times A \xrightarrow{s^\flat \times \mathsf{id}}$ and factor out $\mathsf{id} \times t$ to obtain the equivalent diagram

$$\begin{array}{ccc} \Gamma \times \Delta & \xrightarrow{\mathsf{id} \times t} & \Gamma \times A \\ & & {\scriptstyle s^\flat \times \mathsf{id}}\downarrow \quad \searrow^{s} \\ & & B^A \times A \xrightarrow[\epsilon]{} B \end{array}$$

which commutes by definition, because the triangle is exactly the one described in the adjunction property.

For the η-conversion, we demand that the arrow $f : \Gamma \to B^A$ be equal to the arrow $(\epsilon \circ (f \times \mathsf{id}))^\flat : \Gamma \to B^A$. Writing out the adjunction triangle for the arrow $\epsilon \circ (f \times \mathsf{id}) : \Gamma \times A \to B$, we obtain

$$\begin{array}{ccc} B & \xleftarrow{\epsilon} & B^A \times A \\ {\scriptstyle \epsilon}\uparrow & & \uparrow{\scriptstyle (\epsilon \circ (f \times \mathsf{id}))^\flat \times \mathsf{id}} \\ B^A \times A & \xleftarrow[f \times \mathsf{id}]{} & \Gamma \times A \end{array}$$

By the adjunction property, the arrow $(\epsilon \circ (f \times \mathsf{id}))^\flat : \Gamma \to B^A$ is the unique arrow that makes the square commute. But since we can replace it with the arrow f and obtain a commuting diagram, we conclude that the two arrows are equal.

As such, we are able to interpret types, programs and equations of the simply typed λ-calculus in Cartesian closed categories.

3.2 Models of linear programs

When modeling linear programs, we limit ourselves to programs containing only linear variables — that is, we do not attempt to give interpretations to the !I and !E rules, which only make sense in the presence of intuitionistic variables, as the difficulty of this problem is far

beyond the level of a bachelor's thesis. An interested reader may take a look at the survey (de Paiva, 2014).

We stated multiple times that the intuitionistic product corresponds to the linear & product. However, modeling linear contexts as & products is not desirable, because that would require unnecessary structure — specifically, there is no point in requiring arrows $\pi_1 : A \times B \to A$ for all objects A and B when we cannot produce well-typed programs $\langle x : A\rangle, \langle y : B\rangle \vdash s : A$ for all types A and B, and analogously for the second component. Instead, we introduce a more general concept of a **symmetric monoidal category**, which is used for modeling context concatenation and the $\otimes$ product.

A symmetric monoidal category is a category $\mathcal{C}$ with a symmetric monoidal structure, represented by a functor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, called the **tensor product**, a specific object $1_\otimes$ from $\mathcal{C}$, called the **unit object**, four natural isomorphisms and four kinds of commuting diagrams, listed below.

The first natural isomorphism is the **associator** $\alpha_{X,Y,Z} : (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z)$, which represents associativity of the tensor product. Second and third are the **left** and **right unitor**, written $\lambda_X : 1_\otimes \otimes X \to X$ and $\rho_X : X \otimes 1_\otimes \to X$, respectively. These represent the fact that $1_\otimes$ is the left and right unit of the tensor product. The fourth isomorphism is the **braiding** $B_{X,Y} : X \otimes Y \to Y \otimes X$, which provides the symmetry of the tensor product.

The commuting diagrams enforce that the natural isomorphisms obey our intuitive expectations. The first one is the pentagon identity

$$\begin{array}{ccccc}
((X \otimes Y) \otimes Z) \otimes W & \xrightarrow{\alpha_{X\otimes Y,Z,W}} & (X \otimes Y) \otimes (Z \otimes W) & \xrightarrow{\alpha_{X,Y,Z\otimes W}} & X \otimes (Y \otimes (Z \otimes W)) \\
\downarrow{\scriptstyle \alpha_{X,Y,Z} \otimes \mathsf{id}} & & & & \uparrow{\scriptstyle \mathsf{id} \otimes \alpha_{Y,Z,W}} \\
(X \otimes (Y \otimes Z)) \otimes W & & \xrightarrow{\qquad\qquad \alpha_{X,Y\otimes Z,W} \qquad\qquad} & & X \otimes ((Y \otimes Z) \otimes W)
\end{array}$$

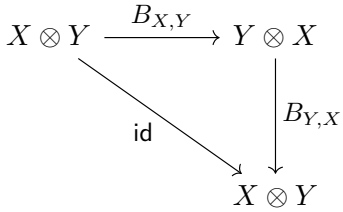
followed by the triangle identity

$$\begin{array}{ccc}
(X \otimes 1_\otimes) \otimes Y & \xrightarrow{\alpha_{X,1_\otimes,Y}} & X \otimes (1_\otimes \otimes Y) \\
& {\scriptstyle \rho_X \otimes \mathsf{id}} \searrow \quad \swarrow {\scriptstyle \mathsf{id} \otimes \lambda_Y} & \\
& X \otimes Y &
\end{array}$$

then the hexagon identity

$$\begin{array}{ccccc}
(X \otimes Y) \otimes Z & \xrightarrow{\alpha_{X,Y,Z}} & X \otimes (Y \otimes Z) & \xrightarrow{B_{X,Y\otimes Z}} & (Y \otimes Z) \otimes X \\
\downarrow{\scriptstyle B_{X,Y} \times Z} & & & & \downarrow{\scriptstyle \alpha_{Y,Z,X}} \\
(Y \otimes X) \otimes Z & \xrightarrow{\alpha_{Y,X,Z}} & Y \otimes (X \otimes Z) & \xrightarrow{\mathsf{id} \otimes B_{X,Z}} & Y \otimes (Z \otimes X)
\end{array}$$

and finally the symmetry condition



holding for all objects X, Y, Z and W of $\mathcal{C}$.

A symmetric monoidal category gives us enough structure to express both context concatenation in LLC (in the same way that Cartesian products interpret context concatenation in STLC) and the $\otimes$ product.

We begin with the linear identity axiom, which is interpreted by the identity arrow

$$\frac{}{A \xrightarrow{\mathsf{id}_A} A}\,\langle \mathrm{Id} \rangle$$

Programs using only linear variables are closed under substitution, because there is no risk of performing a substitution resulting in a linear variable being used multiple times — that would require substituting for intuitionistic variables, which are not present. Therefore, we can interpret substitution with composition, where the program $t[x := s]$ is composed of a term $t : B$ with a free linear variable $x : A$ and possibly others, and a term $s : A$ with possibly free linear variables. The interpretation of t is $\Gamma \otimes A \otimes \Delta \xrightarrow{t} B$, the program s is interpreted as $\Theta \xrightarrow{s} A$, and their composition is $\Gamma \otimes \Theta \otimes \Delta \xrightarrow{\mathsf{id}_\Gamma \otimes s \otimes \mathsf{id}_\Delta} \Gamma \otimes A \otimes \Delta \xrightarrow{t} B$.

From the structural rules, only exchange can be interpreted without intuitionistic variables, and it is satisfied by the symmetry of the tensor product, provided by the isomorphism $B_{A,B}$

$$\frac{\Gamma \otimes A \otimes B \otimes \Delta \xrightarrow{s} C}{\Gamma \otimes B \otimes A \otimes \Delta \underset{B_{A,B}}{\cong} \Gamma \otimes A \otimes B \otimes \Delta \xrightarrow{s} C}\,\mathrm{Exchange}$$

The tensor product functor is used also as interpretation for the $\otimes$ product. The $\otimes$I rule is obvious from the functorial structure

$$\frac{\Gamma \xrightarrow{s} A \quad \Delta \xrightarrow{t} B}{\Gamma \otimes \Delta \xrightarrow{s \otimes t} A \otimes B}\otimes \mathrm{I}$$

where the premise functor $P_{A,B} : \mathcal{C}^{\mathsf{op}} \times \mathcal{C}^{\mathsf{op}} \to \mathbf{Set}$ is given by $P_{A,B}(\Gamma, \Delta) = \mathsf{hom}(\Gamma, A) \times \mathsf{hom}(\Delta, B)$ and the conclusion functor is given by $C_{A,B}(\Gamma, \Delta) = \mathsf{hom}(\Gamma \otimes \Delta, A \otimes B)$. Naturality of the transformation from $P_{A,B}$ to $C_{A,B}$ again corresponds to the ability to do substitution in the contexts Γ and Δ.

The $\otimes$E rule has a more interesting structure — for once, the premise and conclusion functors take arguments other than ones interpreting variable contexts:

$$\frac{\Gamma \otimes A \otimes B \xrightarrow{s} C \quad \Delta \xrightarrow{t} A \otimes B}{\Gamma \otimes \Delta \xrightarrow{\mathsf{id}_\Gamma \otimes t} \Gamma \otimes A \xrightarrow{s} C}\otimes \mathrm{E}$$

The corresponding functors take C as one of their parameters — the premise functor $P_{A,B} : \mathcal{C}^{\mathsf{op}} \times \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathbf{Set}$ is defined by $P_{A,B}(\Gamma, \Delta, C) = \mathsf{hom}(\Gamma \otimes A \otimes B, C) \times \mathsf{hom}(\Delta, A \otimes B)$, and the conclusion functor is defined by $C_{A,B}(\Gamma, \Delta, C) = \mathsf{hom}(\Gamma \otimes \Delta, C)$. Chasing an element of $P_{A,B}(\Gamma, \Delta, C)$ through the diagram of naturality in C, we obtain the equation $(\mathsf{case}\, t\, \mathsf{of}\, |x, y|\, \mathsf{in}\, f[u := s]) \equiv f[u := (\mathsf{case}\, t\, \mathsf{of}\, |x, y|\, \mathsf{in}\, s)]$, which is one of the commuting conversions. Note that this substitution is performed only on programs that only contain linear variables, so the substitution corresponds to filling a term-with-holes that was used without the constructs $(\mathsf{case}\, C[\_]\, \mathsf{of}\, !x\, \mathsf{in}\, s)$ and $(\mathsf{case}\, s\, \mathsf{of}\, !x\, \mathsf{in}\, C[\_])$. Therefore naturality in C recovers the commuting conversion.

For interpreting linear functions, we need a structure similar to exponents in Cartesian closed categories. There, the exponent functors are right adjoint to the Cartesian product, because that is how context concatenation in STLC is interpreted. In LLC, context concatenation is interpreted with the tensor product, thus we require a functor $\_ \multimap \_ : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$, which for any object A in $\mathcal{C}$ gives a right adjoint to the tensor product, written $\_ \otimes A \dashv A \multimap \_$.

Then, the $\multimap$ rules and equations are interpreted exactly the same as the $\to$ rules and equations, except the Cartesian product is replaced with the tensor product. This is possible because none of the $\to$ rules and equations use the projection arrows. Instead they are purely based on the adjunction structure. For example, the $\multimap$ I rule is interpreted by the rule

$$\frac{\Gamma \otimes A \xrightarrow{s} B}{\Gamma \xrightarrow{s^\flat} (A \multimap B)} \multimap \mathrm{I}$$

A symmetric monoidal category equipped with the $\multimap$ functor described above is called a **symmetric monoidal closed category**, and its structure reflects the multiplicative fragment of intuitionistic linear logic — that is only the proofs (or programs) consisting of linear variables and the $\otimes$ and $\multimap$ types.

If we additionally equip the category with finite Cartesian products and include the interpretation of the $\times$ product from STLC, we obtain a Cartesian symmetric monoidal closed category, which is able to interpret all programs from LLC that only use linear variables.

References

Adámek, J., Herrlich, H., & Strecker, G. (1990). *Abstract and concrete categories*. USA: Wiley-Interscience.

Barber, A. (1996). *Dual intuitionistic linear logic* (Tech. Rep.). Edinburgh: University of Edinburgh.

Bauer, A. (2019). *Spartan Martin-Löf type theory.* (Course at University of Ljubljana)

Benton, N., Bierman, G., de Paiva, V., & Hyland, M. (1993a). Linear λ-calculus and categorical models revisited. In *Computer science logic.* Springer Berlin Heidelberg.

Benton, N., Bierman, G., de Paiva, V., & Hyland, M. (1993b). A term calculus for intuitionistic linear logic. In *Proceedings of the international conference on typed lambda calculi and applications (TLCA).* Springer.

Crole, R. (1993). *Categories for types*. Cambridge New York: Cambridge University Press.

de Paiva, V. (2014). Categorical semantics of linear logic for all. In L. C. Pereira, E. H. Haeusler, & V. de Paiva (Eds.), *Advances in natural deduction: A celebration of Dag Prawitz's work* (pp. 181–192). Dordrecht: Springer Netherlands.

Gentzen, G. (1935). Untersuchungen über das logische Schließen. i. *Mathematische Zeitschrift*, *35*, 176–210.

Kleene, S. (1966). *Mathematical logic*. Wiley.

Mac Lane, S. (1998). *Categories for the working mathematician*. New York: Springer.

Paoli, F. (2013). *Substructural logics: A primer*. Springer.

Pfenning, F. (2004). *Automated thoerem proving.* (Course at Carnegie Mellon University)

Sørensen, M. H. B., & Urzyczyn, P. (2006). *Lectures on the Curry-Howard isomorphism*. Amsterdam Boston MA: Elsevier.

Wadler, P. (1993). A taste of linear logic. In A. M. Borzyszkowski & S. Sokołowski (Eds.), *Mathematical foundations of computer science 1993* (pp. 185–210). Berlin, Heidelberg: Springer Berlin Heidelberg.

Zeilberger, N. (2009). *The logical basis of evaluation order and pattern-matching* (Doctoral dissertation, Carnegie Mellon University). Retrieved May 28th, 2021, from `https://www.cs.cmu.edu/~noam/thesis.pdf`