Curry-Howard-Lambek Correspondence

Subashis Chakraborty

August 9,2011

List of Tables

List of Figures

1 Introduction

This paper describes the equivalence of proof sytems (in particular sequent calculi),typed Λ -calculi, and Cartesian closed categories. This is often called Curry-Howard-Lambek correspondence. In order to do this, the paper covers the basic material on proof sytems, typed Λ -calculi and Cartesian closed categories. It defines sequent calculus as the proof system and also gives a short description of natural deduction. Then the paper defines typed- Λ -calculi. Then it proves the correspondence between sequent calculi and typed Λ -calculi which is generally known as Curry-Howard correspondence. After that the paper gives the definition of Cartesian closed category. Then the equivalence of Cartesian closed categories and typed Λ -calculi and proof sytems is proved.

2 Proof Sytem

The Curry-Howard correspondence,in general, is a correspondence between formal proof calculus and type systems for models of computation.It can be divided into two correspondences. One at the level of formulas and types that is independent of which particular system or model of computation is considered, and one at the level of proofs and programs which, this time, is specific to the particular choice of proof system and model of computation considered.

At the level of formulas and types, the correspondence says that implication behaves the same as a function type, conjunction as a "product" type(this may be called a tuple, a struct,a list, or some other term depending on the language), disjunction as a sum type(this type may be called a union), the false formula as the empty type and the true formula as the singleton type(whose sole member is null object). Quantifiers correspond to dependent products or sums. This is summarized in the following $Table1$. At the level of proof systems and model of computations, the correspondence mainly shows the identity of structure, first

Logic side	Programming side
universal quantification	dependent product type
existential quantification	dependent sum type
implication	function type
conjunction	product type
disjunction	sum type
true formula	unit type
false formula	bottom type

Table 1: General Formulation

between some particular formulationsof systems known as Hilbert-style deduction system and combinatory logic, and , secondly, between some particular formulations of systems known as natural deduction and lambda calculus.

Logic side	Programming side
Hilbert-style deduction system	type system for combinatory logic
Natural deduction	type system for Λ-calculi
Sequent calculi (Herbelin Version)	type system for $\bar{\Lambda}$-calculi

Table 2: Correspondence between Proof System and Programming Side

2.1 Natural Deduction

In logic and proof theory, natural deduction is a kind of proof calculus in which logical reasoning is expressed by inference rules closely related to the "natural" way of reasoning. This contrasts with the axiomatic systems which instead use axioms as much as possible to express the logical laws of deductive reasoning.

2.2 Gentzen's LJ Sequent calculus

Natural deduction in its modern form was independently proposed by the German mathematician Gentzen in 1935, in a dissertation delivered to the faculty of mathematical sciences of the university of Gttingen.Gentzen was motivated by a desire to establish the consistency of number theory, and he found immediate use for his natural deduction calculus. He was nevertheless dissatisfied with the complexity of his proofs, and in 1938 gave a new consistency proof using his sequent calculus.In proof theory and mathematical logic, sequent calculus is a family of formal systems sharing a certain style of inference and certain formal properties. The first sequent calculi, systems LK and LJ, were introduced by Gerhard Gentzen in 1934 as a tool for studying natural deduction in first-order logic (in classical and intuitionistic versions, respectively).

The main idea in this Gentzen's Sequent calculus is that, a sequent $\Gamma \vdash \Delta$ consists of two finite multi-sets Γ and Δ of formula, and that rater than having introduction and elimination rules, we have rules introducing a connective on the left or on the right of a sequent. A first version of such a system for classical propositional logic is given next. In this rules Γ and Δ stand for possibly empty finite multisets of propositions. The inference rules for this Sequent calculus is giiven in the $Figure1$.

Note the perfect symmetry of the left and right rules. If one wants to deal with the extended language containing also $\bot$, one needs to add the axiom $\bot, \Gamma \vdash \Delta$.

One might be puzzled and even concerned about the presence of the contraction rule. Indeed, one might wonder whether the presence of this rule will not cause provability to be undecidable. This would certainly

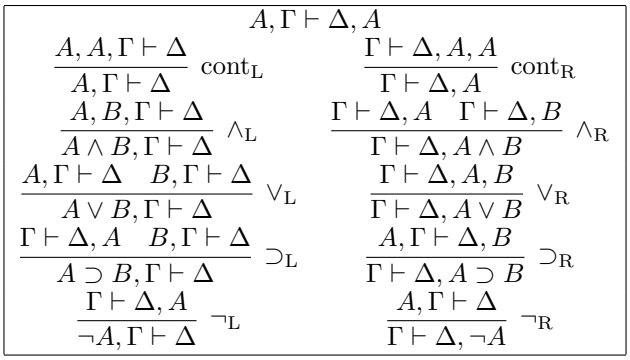


Figure 1: Gentzen's LJ Sequent calculus

be quite bad, since we are only dealing with propositions! Fortunately, it can be shown that the contraction rule is redundant for classical propositional logic. But then, why include it in the first place? The main reason is that it cannot be dispensed with in intuitionistic logic, or in the case of quantified formula. Since we would like to view intuitionistic logic as a subsystem of classical logic, we cannot eliminate the contraction rule from the presentation of classical systems. Another important reason is that the contraction rule plays an important role in cut elimination. Although it is possible to hide it by dealing with sequents viewed as pairs of sets rather than multisets, we prefer to deal with it explicitly.Finally, the contraction rule plays a crucial role in linear logic, and in the understanding of the correspondence between proofs and computation, in particular strict versus lazy evaluation.

2.3 Herbelin LJT Sequent calculus

2.3.1 Formulas and sequents

Definition 2.1. *The set of* **formulas** *is an inductive structure defined by the following grammar where X ranges over V_F.*

$$A ::= X | A \longrightarrow A$$

We use the letters $A, B, C, ...$ to denote formulas.

Definition 2.2. *An* **environment** *is a set of formulas. It may occur several times the same formula in an environment but each time with a different associated name.*

We use the letters $\Gamma, \Delta, \Omega, ...$ to denote sets of named formulas.

Definition 2.3. *In a sequent, written under the form $\Gamma; \Pi \vdash A$, the place on the right of the semicolon is called the* **stoup**. *The stoup may be either empty or containing a single formula. Π is such a notation to express "nothing or a formula".*

Thus, there are two kinds of sequents in LJT. Those with an empty stoup and those with a non empty stoup.The inference rules of LJT are given in the *Figure* 2.

The cut elimination rules for Herbelin LJT Sequent calculus is given in *Figure*3.

3 Typed λ -Calculus

The λ -calculus was one of the first descriptions produced of computable functions. It grew out of an attempt by Alonso Church to provide a foundation for mathematics using functions as a basic building block. The

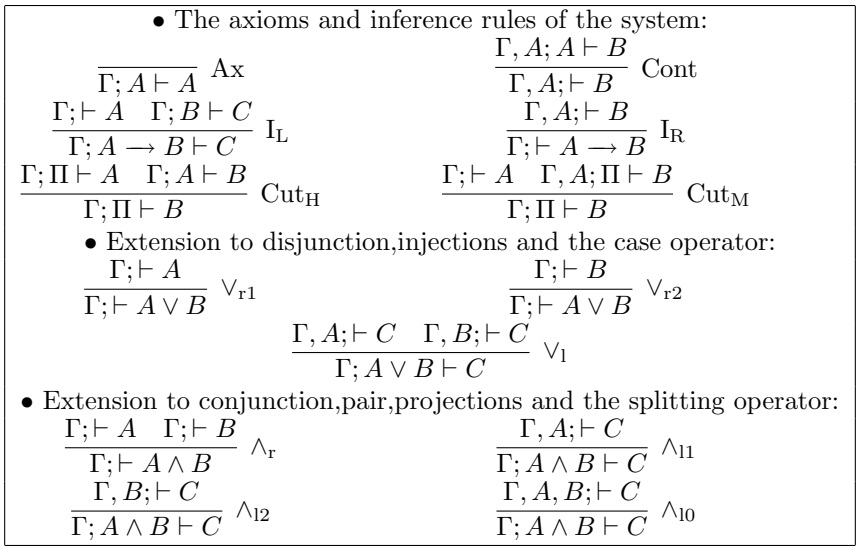


Figure 2: Axioms and Inference rules

atempt failed due to a form of Russell's paradox(which gave fixed point combinators). Church decided to extract a part of the calculus which worked perfectly well even if it did not achieve his original objective.

The λ -calculus is an algebra with a binary "application" operation and an "abstraction" operator. The application is sometimes written $f \bullet x$ (pronounced "f applied to x") when we wish to emphasize the presence of a binary operation.A typed lambda calculus is a typed formalism that uses the lambda-symbol () to denote anonymous function abstraction. In this context, types are usually objects of a syntactic nature that are assigned to lambda terms; the exact nature of a type depends on the calculus considered (see kinds below). From a certain point of view, typed lambda calculi can be seen as refinements of the untyped lambda calculus but from another point of view, they can also be considered the more fundamental theory and untyped lambda calculus a special case with only one type.

Typed lambda calculus are foundational programming languages and are the base of typed functional programming languages such as ML and Haskell and, more indirectly, typed imperative programming languages. Typed lambda calculi play an important role in the design of type systems for programming languages; here typability usually captures desirable properties of the program, e.g. the program will not cause a memory access violation. Typed lambda calculus are closely related to mathematical logic and proof theory via the CurryHoward isomorphism and they can be considered as the internal language of classes of categories, e.g. the simply typed lambda calculus is the language of Cartesian closed categories (CCCs) (via Curry-Howard-Lambek correspondence).

The terms in the λ -calculus are formed by the rules given in *Figure*4.

3.1 The $\bar{\lambda}$-calculus

3.1.1 The $\bar{\lambda}$-expressions

$\bar{\lambda}$-expressions are comprised of $\bar{\lambda}$-terms and lists of $\bar{\lambda}$-terms. Actually, lists of $\bar{\lambda}$-terms mean lists of arguments. These are dependent to each other.

Definition 3.1.

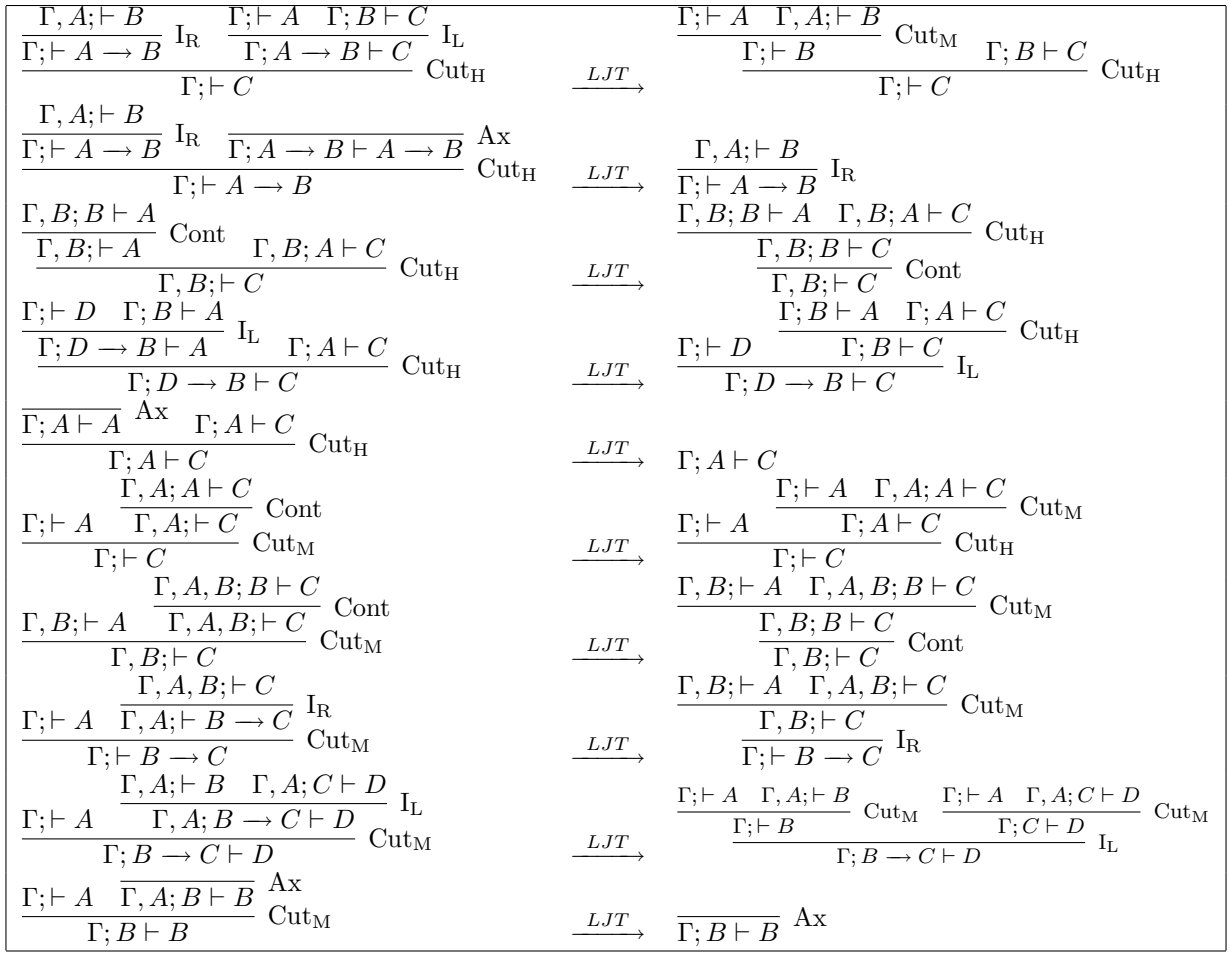


Figure 3: Rules for Cut-elimination

The set of $\bar{\lambda}$-expressions are defined by

Term: $t := (x\ l)|(\lambda x.t)|(t \circ l)|((t/x) \triangleright t)$
Lists of arguments: $l := [\,]|[t :: l]|(l\ @\ l)|l \bullet [x := t]$

Here the letters $t, u, v, ...$ are used for terms and the syntactic expressions $l, l', ...$ are used for argument lists. [] stands for the empty argument list. and $[t :: l]$ stands for the term t to the list of arguments l, while $(l@l')$ stands for the explicit concatenation of the lists l and l' of arguments.

If we consider case operator, disjunction , injection , conjunction, pair, projections and the splitting operator in this calculus then the new set of terms are defind by

Term: $t := (x\ l)|(\lambda x.t)|(t \circ l)|((t/x) \triangleright t)|\sigma_1(t)|\sigma_2(t)|\langle t, t\rangle$
Lists of arguments: $l := [\,]|[t :: l]|(l\ @\ l)|l \bullet [x := t]|[\langle (x)t|(x)t\rangle]|[\langle x, x\rangle t]|[\pi_1 :: l]|[\pi_2 :: l]$

The argument list $[\langle (x)t|(y)u\rangle]$ stands for a case operator. If we see it as the applicative context $(.[\langle (x)t|(y)u\rangle])$, then we can adopt the more expressive notation

$$\frac{x \;\; Variable}{x \;\; Term}\ \text{Var}$$

$$\frac{M \;\; Term \quad N \;\; Term}{MN \;\; Term}\ \text{app}$$

$$\frac{M \;\; Term \quad x \;\; Variable}{\lambda x.M \;\; Term}\ \text{abst}$$

Figure 4: Rules for Term Formation in λ-calculus

$$case\ .\ of \begin{cases} \sigma_1(x) \longrightarrow t \\ \sigma_2(y) \longrightarrow u \end{cases}$$

3.1.2 Reduction rules

• General rules:

$$\begin{array}{rcl}
(\lambda x.u\ [v :: l]) & \xrightarrow{r} & (((v/x) \triangleright u)\ l) \\
(\lambda x.u\ []) & \xrightarrow{r} & \lambda x.u \\
((x\ l)\ l') & \xrightarrow{r} & ((x\ (l\ @\ l')) \\
[u :: l]\ @\ l' & \xrightarrow{r} & [u :: (l\ @\ l')] \\
[\,]\ @\ l' & \xrightarrow{r} & l' \\
(v/x) \triangleright (x\ l) & \xrightarrow{r} & (v\ l \bullet [\ x\ :=\ v]) \\
(v/x) \triangleright (y\ l) & \xrightarrow{r} & (y\ l \bullet [\ x\ :=\ v]) \\
(v/x) \triangleright (\lambda y.u) & \xrightarrow{r} & \lambda y.((v/x) \triangleright u) \\
[\,] \bullet [\ x\ :=\ v] & \xrightarrow{r} & [\,] \\
[u :: l] \bullet [\ x\ :=\ v] & \xrightarrow{r} & [((v/x) \triangleright u) :: l \bullet [\ x\ :=\ v]]
\end{array}$$

• Extension to disjunction,injections and the case operator:

$$\begin{array}{rcl}
(inj_1(t)\ [\langle (x)u | (y)v \rangle]) & \xrightarrow{r} & ((t/x) \triangleright u) \\
(inj_2(t)\ [\langle (x)u | (y)v \rangle]) & \xrightarrow{r} & ((t/y) \triangleright v) \\
(v/z) \triangleright inj_1(t) & \xrightarrow{r} & inj_1((v/z) \triangleright t) \\
(v/z) \triangleright inj_2(t) & \xrightarrow{r} & inj_2((v/z) \triangleright t) \\
[\langle (x)u | (y)v \rangle] \bullet [z := w] & \xrightarrow{r} & [\langle (x)((w/z) \triangleright u) | (y)((w/z) \triangleright v) \rangle] \\
[\langle (x)u | (y)v \rangle]\ @\ l' & \xrightarrow{r} & [\langle (x)(u \circ\ l') | (y)(v \circ\ l') \rangle]
\end{array}$$

• Extension to conjunction,pair,projections and the splitting operator:

$$\begin{array}{rcl}
(\langle t, u \rangle\ [\pi_1 :: l]) & \xrightarrow{r} & (t \circ l) \\
(\langle t, u \rangle\ [\pi_2 :: l]) & \xrightarrow{r} & (u \circ l) \\
(\langle t, u \rangle\ [< x, y >\ v]) & \xrightarrow{r} & ((u/y) \triangleright ((t/x) \triangleright v)) \\
(v/z) \triangleright \langle t, u \rangle & \xrightarrow{r} & \langle ((v/z) \triangleright t), ((v/z) \triangleright u) \rangle \\
[\pi_1 :: l] \bullet [z := v] & \xrightarrow{r} & [\pi_1 :: l \bullet [z := v]] \\
[\pi_2 :: l] \bullet [z := v] & \xrightarrow{r} & [\pi_2 :: l \bullet [z := v]] \\
[\langle x, y \rangle v] \bullet [z := v] & \xrightarrow{r} & [\langle x, y \rangle (v[z := v])] \\
[\pi_1 :: l] @ l' & \xrightarrow{r} & [\pi_1 :: l @ l'] \\
[\pi_2 :: l] @ l' & \xrightarrow{r} & [\pi_2 :: l @ l'] \\
[\langle x, y \rangle v] @ l' & \xrightarrow{r} & [\langle x, y \langle (vl')]
\end{array}$$

- Extension to quantifiers and to product and sigma operators:

$$\begin{array}{rcl}
(\lambda r.u[a :: l]) & \xrightarrow{r} & (u'l) \\
(\langle a,t\rangle\ [\langle r,x\rangle u]) & \xrightarrow{r} & (u\ l) \\
(\lambda r.u)[z := v] & \xrightarrow{r} & \lambda r.(u[z := v]) \\
\langle a,t\rangle\ [z := v] & \xrightarrow{r} & \langle a, t\ [z := v]\rangle \\
[a :: l][z := v] & \xrightarrow{r} & [a :: l[z := v]] \\
([\langle r,x\rangle u])[z := v] & \xrightarrow{r} & [\langle r,x\rangle(u[z := v])] \\
[a :: l]@l' & \xrightarrow{r} & [a :: (l@l')] \\
[\langle r,x\rangle u]@l' & \xrightarrow{r} & [\langle r,x\rangle(ul')]
\end{array}$$

4 The assignment of proofs in LJT by $\bar{\lambda}$-expressions

Definition 4.1. *A* **declaration***, written under the form $x : A$ is the pair of a term variable name and a formula.*

Definition 4.2. *Here the word* **environment** *is kept, to denote a sequence of declarations and we use the same letters Γ, Δ, Ω as before to denote them.*

Definition 4.3. *An* **applicative context** *is a list of arguments written under the form $(.l)$ where $.$ is a special notational symbol. Also, We call hole declaration a formula written under the form $. : A$.*

Definition 4.4. *A* **judgement** *is something of the form $\Gamma; \Pi \vdash t : A$. In this writing Π is either nothing, in which case t is a term, or a hole declaration in which case t is an applicative context.*

The formation rules for $\bar{\lambda}$-calculus are defined in $Figure 5$.

5 Curry-Howard correspondence

The **Curry-Howard correspondence** is the direct relationship between computer programs and proofs in constructive mathematics. It is a generalization of a syntactic analogy between systems of formal logic and computational calculus that was first discovered by the American mathematician Haskell Curry and logician William Alvin Howard.

At the very beginning, the **Curry-Howard correspondence** is

1) the observation in 1934 by Curry that the types of the combinators could be seen as axiom-schemes for intituitionistic implicational logic.

2) the observation in 1958 by Curry that a certain kind of proof system, referred to as Hilbert-style deduction systems, coincides on some fragment to the typed fragment of a standard model of computation known as combinatory logic.

3) the observation in 1969 by Howard that another, more "high-level" proof system, referred to as natural deduction, can be directly interpreted in its intutionistic version as a typed variant of the model of computation known as lambda calculus.

In-other words, the Curry-Howard correspondence is the observation that two families of formalisms which had seemed unrelated-namely, the proof systems on one hand, and the models of computation on the other-were, in the two examples of Curry and Howard, infact structurally the same kinds of objects.

If one now abstracts on the pecularities of this or that formalism, the immediate generalization is the

•Applicatiive context formation rules

$$\frac{}{\Gamma;.:A\vdash(.\ [\,]):A}\ \mathrm{Ax}$$

$$\frac{\Gamma;\vdash u:A\quad\Gamma;.:B\vdash(.\ l):C}{\Gamma;.:A\longrightarrow B\vdash(.[u::l]):C}\ \mathrm{I_L}$$

$$\frac{\Gamma;.:C\vdash(.\ l):A\quad\Gamma;.:A\vdash(.\ l'):B}{\Gamma;.:C\vdash(.\ (l\ @\ l')):B}\ \mathrm{Cut_H}$$

$$\frac{\Gamma;\vdash u:A\quad\Gamma,x:A;.:C\vdash(.\ l):B}{\Gamma;.:C\vdash(.\ l\bullet[x:=u]):B}\ \mathrm{Cut_M}$$

•Term Formation rules

$$\frac{\Gamma,x:A;.:A\vdash(.\ l):B}{\Gamma,x:A;\vdash(x\ l):B}\ \mathrm{Cont}$$

$$\frac{\Gamma,x:A;\vdash u:B}{\Gamma;\vdash\lambda x.u:A\longrightarrow B}\ \mathrm{I_R}$$

$$\frac{\Gamma;\vdash u:A\quad\Gamma;.:A\vdash(.\ l):B}{\Gamma;\vdash(u\circ l):B}\ \mathrm{Cut_H}$$

$$\frac{\Gamma;\vdash u:A\quad\Gamma,x:A;\vdash v:B}{\Gamma;\vdash(u/x)\triangleright v:B}\ \mathrm{Cut_M}$$

•Extension to disjunction,injections and the case operator:

$$\frac{\Gamma;\vdash u:\ A}{\Gamma;\vdash\sigma_1(u):A\vee B}\ \vee_{\mathrm{r1}}$$

$$\frac{\Gamma;\vdash v:\ B}{\Gamma;\vdash\sigma_2(v):A\vee B}\ \vee_{\mathrm{r2}}$$

$$\frac{\Gamma,x:\ A;\vdash u:\ C\quad\Gamma,y:\ B;\vdash v:\ C}{\Gamma;.:A\vee B\vdash\Delta;(.[\langle(x)u|(y)v\rangle]:C}\ \vee_{\mathrm{l}}$$

• Extension to conjunction,pair,projections and the splitting operator:

$$\frac{\Gamma;\vdash u:A\quad\Gamma;\vdash v:B}{\Gamma;\vdash\langle u,v\rangle:A\wedge B}\ \wedge_{\mathrm{r}}$$

$$\frac{\Gamma,.:A;\vdash(.l):C}{\Gamma;.:A\wedge B\vdash\Delta;(.[\pi_1::l]):C}\ \wedge_{\mathrm{l1}}$$

$$\frac{\Gamma,.:B;\vdash(.l):C}{\Gamma;.:A\wedge B\vdash\Delta;(.[\pi_2::l]):C}\ \wedge_{\mathrm{l2}}$$

$$\frac{\Gamma,x:A,y:B;\vdash u:C}{\Gamma;.:A\wedge B\vdash\Delta;(.[\langle x,y\rangle u]):C}\ \wedge_{\mathrm{l0}}$$

Figure 5: Formation Rules in $\bar{\lambda}$-calculus

Intutionistic implicational natural deduction	Lambda calculus rules
$\frac{}{\Gamma_1,\alpha,\Gamma_2\vdash\alpha}\mathrm{Ax}$	$\frac{}{\Gamma_1,x{:}\alpha,\Gamma_2\vdash x{:}\alpha}$
$\frac{}{\Gamma,\alpha\vdash\beta}\longrightarrow I$	$\frac{\Gamma,x{:}\alpha\vdash t{:}\beta}{\Gamma\vdash\lambda x.t{:}\alpha\longrightarrow\beta}$
$\frac{\Gamma\vdash\alpha\longrightarrow\beta\Gamma\vdash\alpha}{\Gamma\vdash\beta}\longrightarrow E$	$\frac{\Gamma\vdash t{:}\alpha\longrightarrow\beta\Gamma\vdash u{:}\alpha}{\Gamma\vdash tu{:}\beta}$

Table 3: Correspondence between natural deduction and lambda calculus

following claim: *a proof is a program, the formula it provides is a type for the program.* Most informally, this can be seen as an analogy that states that the return type of a function (i.e., the type of values returned by a function) is analogous to a logical theorem, subject to hypotheses corresponding to the types of the argument values passed to the function; and that the program to compute that function is analogous to a proof of that theorem, this sets a form of logic programming on a rigorous foundation: *proofs can be represented as programs, and especially as lambda terms, or proofs can be run.*

The correspondence has been the starting point of a large spectrum of new research after its discovery, leading in particular to a new class of formal systems designed to act both as a proof system and as a typed functional programming language.

Definition 5.1. *The set of* **formulas** *is an inductive structure defined by the following grammar where X ranges over V_F.*

$$A ::= X | A \longrightarrow A$$

We use the letters $A, B, C, ...$ to denote formulas.

The converse direction is to use a program to extract a proof, given its correctness - an area of research which is closely releated to proof-carrying code. this is only feasible if the programming language the program is written for is very richly typed: the development of such tyep systems has partly motivated by the wish to make the Curry-Howard correspondence practically relevant.

5.1 Correspondence between natural deduction and lambda calculus

After Curry emphasized the syntactic correspondence between Hilbert-style deduction and combinatory logic,Howard made explicit in 1969 a syntactic analogy between the programs of simply-typed lambda calculus and the proof of natural deduction. Below, the left-hand side formalizes intutionistic natural deduction as a calculus of sequents(the use of sequents is standard in discussions of the Curry-Howard isomorphism as it allows the deductioin rules to be stated more cleanly) with implicit weakening and the right hand side shows the typing rules of lambda calculus. in the left-hand side,Γ, $\Gamma 1$ and $\Gamma 2$ denote ordered sequences of formulas while in the right hand side, they denote sequences of named formulas with all names different.The correspondence between Intutionistic implicational natural deduction and lambda calculus are shown in $Table 3$.

To paraphrase the correspondence, proving $\Gamma \vdash \alpha$ means having a program that, given values with the types listed in Γ,manufactures an object of type α. An axiom corresponds to the instroduction of a new variable with a new, unconstrained type, the $\longrightarrow I$ rule corresponds to function abstraction and $\longrightarrow E$ rule corresponds to function application. Observe that the correspondence is not exact if the context Γ is taken to be a set of formulas as e.g., the λ -terms $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$ of type $\alpha \longrightarrow \alpha \longrightarrow \alpha$ would not be distinguished in the correspondence.

Example 5.1. Proof and Program correspondence: *Here a proof of $(\beta \longrightarrow \alpha) \longrightarrow (\gamma \longrightarrow \beta) \longrightarrow \gamma \longrightarrow \alpha$ in natural deduction and show how it can be interpreted as the λ - expression $\lambda a.\lambda b.\lambda g.(a(bg))$ of type $(\beta \longrightarrow \alpha) \longrightarrow (\gamma \longrightarrow \beta) \longrightarrow \gamma \longrightarrow \alpha$.*

$$\dfrac{\dfrac{}{a:\beta\longrightarrow\alpha, b:\gamma\longrightarrow\beta, g:\gamma\vdash a:\beta\longrightarrow\alpha}\quad\dfrac{a:\beta\longrightarrow\alpha, b:\gamma\longrightarrow\beta, g:\gamma\vdash b:\gamma\longrightarrow\beta \qquad a:\beta\longrightarrow\alpha, b:\gamma\longrightarrow\beta, g:\gamma\vdash g:\gamma}{a:\beta\longrightarrow\alpha, b:\gamma\longrightarrow\beta, g:\gamma\vdash b\ g:\beta}}{\dfrac{\dfrac{\dfrac{a:\beta\longrightarrow\alpha, b:\gamma\longrightarrow\beta, g:\gamma\vdash a\ (b\ g):\alpha}{a:\beta\longrightarrow\alpha, b:\gamma\longrightarrow\beta\vdash\lambda g.a\ (b\ g):\gamma\longrightarrow\alpha}}{a:\beta\longrightarrow\alpha\vdash\lambda b.\lambda g.a\ (b\ g):(\alpha\longrightarrow\beta)\longrightarrow\gamma\longrightarrow\alpha}}{\vdash\lambda a.\lambda b.\lambda g.a\ (b\ g):(\beta\longrightarrow\alpha)\longrightarrow(\alpha\longrightarrow\beta)\longrightarrow\gamma\longrightarrow\alpha}$$

5.2 Correspondence between Herbelin LJT Sequent calculus and $\bar{\lambda}$-calculus

5.2.1 Correspondence between Cut elimination and reduction rules

Each cut elimination rule corresponds to the reduction rules described in the $\bar{\lambda}$-calculus. The first cut elimination rule is

$$\dfrac{\dfrac{\Gamma, A;\vdash B}{\Gamma;\vdash A\longrightarrow B}\ \mathrm{I_R}\quad\dfrac{\Gamma;\vdash A\quad\Gamma;B\vdash C}{\Gamma;A\longrightarrow B\vdash C}\ \mathrm{I_L}}{\Gamma;\vdash C}\ \mathrm{Cut_H}\quad\xrightarrow{LJT}\quad\dfrac{\dfrac{\Gamma;\vdash A\quad\Gamma, A;\vdash B}{\Gamma;\vdash B}\ \mathrm{Cut_M}\quad\Gamma;B\vdash C}{\Gamma;\vdash C}\ \mathrm{Cut_H}$$

After assignemnt of proofs in calculus:

$$\dfrac{\dfrac{\Gamma, a:A;\vdash t:B}{\Gamma;\vdash(\lambda a.t):A\longrightarrow B}\ \mathrm{I_R}\quad\dfrac{\Gamma;\vdash t_1:A\quad\Gamma;.:B\vdash(.l):C}{\Gamma;.:A\longrightarrow B\vdash(.[t_1::l]):C}\ \mathrm{I_L}}{\Gamma;\vdash(\lambda a.t)[t_1::l]:C}\ \mathrm{Cut_H}\quad\xrightarrow{LJT}\quad\dfrac{\dfrac{\Gamma;\vdash t_1:A\quad\Gamma, a:A;\vdash t:B}{\Gamma;\vdash t[a:=t_1]:B}\ \mathrm{Cut_M}\quad\Gamma;.:B\vdash(.l):C}{\Gamma;\vdash(t[a:=t_1]l):C}\ \mathrm{Cut_H}$$

After assigning of proofs in calculus, it corresponds to the first reduction rule described in the 3.1.2.

$$(\lambda x.u\ [v::l])\quad\xrightarrow{r}\quad(u[x:=v]\ l)$$

5.2.2 Proof Example

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, A \longrightarrow B, B \longrightarrow C; A \vdash A}\,\mathrm{Ax}}{A, A \longrightarrow B, B \longrightarrow C; \vdash A}\,\mathrm{Cont} \quad \dfrac{}{A, A \longrightarrow B, B \longrightarrow C; B \vdash B}\,\mathrm{Ax}}{A, A \longrightarrow B, B \longrightarrow C; A \longrightarrow B \vdash B}\,\mathrm{I_L}}{A, A \longrightarrow B, B \longrightarrow C; \vdash B}\,\mathrm{Cont} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{A, A \longrightarrow B, B, B \longrightarrow C; B \vdash B}\,\mathrm{Ax}}{A, A \longrightarrow B, B, B \longrightarrow C; \vdash B}\,\mathrm{Cont} \quad \dfrac{}{A, A \longrightarrow B, B, B \longrightarrow C; C \vdash C}\,\mathrm{Ax}}{A, A \longrightarrow B, B, B \longrightarrow C; B \longrightarrow C \vdash C}\,\mathrm{I_L}}{A, A \longrightarrow B, B, B \longrightarrow C; \vdash C}\,\mathrm{Cont}}{A, A \longrightarrow B, B \longrightarrow C; \vdash C}\,\mathrm{Cut_M}}{A, A \to B, B \longrightarrow C; \vdash C}\,\mathrm{I_R}$$

$$\dfrac{A \longrightarrow B, B \longrightarrow C; \vdash A \longrightarrow C}$$

5.2.3 Proof Example with Type Assignment

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a : A, x : A \longrightarrow B, y : B \longrightarrow C; . : A \vdash .[] : A}\,\mathrm{Ax}}{a : A, x : A \longrightarrow B, y : B \longrightarrow C; \vdash a[] : A}\,\mathrm{Cont} \quad \dfrac{}{a : A, x : A \longrightarrow B, y : B \longrightarrow C; . : B \vdash .[] : B}\,\mathrm{Ax}}{a : A, x : A \longrightarrow B, y : B \longrightarrow C; . : A \longrightarrow B \vdash [a[]] : B}\,\mathrm{I_L}}{a : A, x : A \longrightarrow B, y : B \longrightarrow C; \vdash x[[a]] : B}\,\mathrm{Cont} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{a : A, x : A \longrightarrow B, b : B, y : B \longrightarrow C; . : B \vdash .[] : B}\,\mathrm{Ax}}{a : A, x : A \longrightarrow B, b : B, y : B \longrightarrow C; \vdash b[] : B}\,\mathrm{Cont} \quad \dfrac{}{a : A, x : A \longrightarrow B, b : B, y : B \longrightarrow C; . : C \vdash .[] : C}\,\mathrm{Ax}}{a : A, x : A \longrightarrow B, b : B, y : B \longrightarrow C; . : B \longrightarrow C \vdash [b[]] : C}\,\mathrm{I_L}}{a : A, x : A \longrightarrow B, b : B, y : B \longrightarrow C; \vdash y[b[]] : C}\,\mathrm{Cont}}{a : A, x : A \longrightarrow B, y : B \longrightarrow C; \vdash ((x[a[]]/b) \triangleright (y[b[]])) : C}\,\mathrm{Cut_M}}{x : A \longrightarrow B, y : B \longrightarrow C; \vdash \lambda a.((x[a[]]/b) \triangleright (y[b[]])) : A \longrightarrow C}\,\mathrm{I_R}$$

6 Cartesian closed categories

Definition 6.1. *A category G is called a* **cartesian closed category** *if it satisfies the following:*

[CCC-1] *There is a terminal object 1.*

[CCC-2] *Each pair of objects A and B of C has a product $A \times B$ with projections $p_1 : A \times B \longrightarrow A$ and $p_2 : A \times B \longrightarrow B$.*

[CCC-3] *For every pair of objects A and B, there is an object $[A \longrightarrow B]$ and an arrow eval: $[A \longrightarrow B] \times A \longrightarrow B$ with the property that for any arrow $f : C \times A \longrightarrow B$, there is a unique arrow $\lambda f : C \longrightarrow [A \longrightarrow B]$ such that the composite*

$$C \times A \xrightarrow{\lambda f \times A} [A \longrightarrow B] \times A \xrightarrow{eval} B$$

is f.

Example 6.1. *The example of a cartesian closed category is the category of sets. For any sets A and B, $[A \longrightarrow B]$ is the set of functions from A to B, $eval^B_A$ is the evaluation or apply function. In the case of* **Set**,*$[A \longrightarrow B]$ is $Hom_{Set}(A, B)$.*

7 Curry-Howard-Lambek correspondence

Joachim Lambek showed in the early 1970s that the proofs of intuitionstic propositional logic and the combinators of typed combinatory logic share a common equitional theory which is the one of cartesian closed categories. The three way isomorphism between intuitionistic logic, typed lambda calculus and cartesian closed category is called Curry-Howard-Lambek correspondence.

Given a typed λ -calculus L, the objects of the category C(L) are the types of L. An arrow from an object A to an object B is an equivalence class of terms of type B with one free variable of type A (which need not actually occur in the terms).

The equivalence relation is the least reflexive, symmetric,transitive relation induced by saying that two such terms $\emptyset(x)$ and $\psi(y)$ are equvalent if $\emptyset$ and ψ are both of the same type, x and y are both of the same type, x and y are both of the same type, x is substitutionable for y in ψ, and $\emptyset(x) =_{\{x\}} \psi(x)$, where $\psi(x)$ is obtained from $\psi(y)$ by substituting x for every occurence of y.

The reason we need equivalence classes is that any two variables of the same type must correspond to the same arrow, the identity, of the object to itself. If $\lambda_{x \epsilon A} x : 1 \longrightarrow [A \longrightarrow A]$ is to name of the identity arrow of A for any variable $x \epsilon A$, as is intuitively evident, then the arrow corresponding to a variable x of type A must be the identity of A.

The equivalence relation of also akes two terms containing a variable fo type 1 equivalent(because of TL-10), thus ensuring that 1 will be a terminal object of the category.

Suppose $\emptyset$ is a term of type B with at most one free variable x of type A and ψ is a term of type C with atmost one free variable y of type B. Note that by replacing, if necessary, x by a variable that is not bound in ψ, we can assume that $\emptyset$ is substitutable for y in ψ. We then define the composite of the corresponding arrows to be the arrow which is the equivalence class of the term $\psi(\emptyset)$ obtained by substituting ψ for x in ψ.

Example 7.1. *Here a calculation can be exhibited that verifies one of the properties of a cartesian closed category as an example of how the definition of C(L) works.*

Let L be a typed λ-calculus. Define

$$\Lambda : Hom_{C(L)}(C \times A \longrightarrow B) \longrightarrow Hom_{C(L)}(C \longrightarrow [A \longrightarrow B])$$

as follows: for $[\emptyset(u)] : C \times A \longrightarrow B$ (so that u is a variable of type $C \times A$), $\Lambda([\emptyset(u)]) = \lambda_x \emptyset((z, x))$,where z is a variable of type C and x is a variable of type A. Define

$$\Gamma : Hom_{C(L)}(C \longrightarrow [A \longrightarrow B]) \longrightarrow Hom_{C(L)}(C \times A \longrightarrow B)$$

as follows: for $[\psi(z)] : C \longrightarrow [A \longrightarrow B], \Gamma([\psi(z)]) = \psi(proj_1 u)`proj_2 u$, where u is a variable of type $C \times A$. Then λ and Γ are inverse functions.

Here is one direction of the verification. The following calculation uses TL-18 and the fact that x does not occur in $\emptyset(z)$ because by definition of arrow in $C(L)$, $\emptyset(z)$ contains only one variable and that is not x.

$$\lambda(\Gamma(\psi(z))) = \Lambda(\psi(proj_1 u)`proj_2 u) = \lambda_x(\psi(proj_1(z, x))`proj_2(z, x)) =_X \lambda_x \psi(z)`x =_X \psi(z)$$

Theorem 7.2. *Let C be a cartesian closed category with internal language L. Then C(L) is a category equivalent to C.*

They define what it means for languages to be equivalent and show that if you start with a typed λ -calculus, construct the corresponding category, and then construct the internal language, the language and the original typed λ -calculus are equivalent. They state this in a more powerful way in the language of adjunctions.

7.1 The cartesian closed category generated by a typed λ-*calculus*

To show that the functor **L**(typed λ-*calculus*) is an equivalence of categories we shall obtain a functor C in the opposite direction.

Given a typed λ-*calculus* L, we construct a cartesian closed category C(L) with weak natural numbers object as follows:

• **Object** The objects of C(L) are the types of L.

• **Arrow** The arrows $A \longrightarrow B$ of $C(L)$ are (equivalence classes of) pairs $(x \epsilon A, \phi(x))$,with x a variable of type B with no free variables other than x. (Think of function $x \mapsto \phi(x)$.)

Equality of arrows is defined by: $(x \epsilon A, \phi(x)) = (x' \epsilon A, \psi(x'))$ if and only if $\phi(x) \stackrel{x}{=} \psi(x)$ holds , where $\stackrel{x}{=}$ abbreviates $\stackrel{\{x\}}{=}$.

• **Identity** The identity arrow $A \longrightarrow A$ is the pair $(x \epsilon A, x)$

• **Composition** The composition of $(x \epsilon A, \phi(x)) : A \longrightarrow B$ and $(y \epsilon B, \psi(y)) : B \longrightarrow C$ is given by $(x \epsilon A, \psi(\phi(x))) : A \longrightarrow C, \psi(x)$ having been substituted for y in $\psi(y)$

The cartesian closed structure of C(L) is obtained as follows:

$$\bigcirc_A \equiv (x \epsilon A, *),$$

$$\pi_{A,B} \equiv (z\epsilon A \times B, \pi(z)),$$
$$\pi'_{A,B} \equiv (z\epsilon A \times B, \pi'(z)),$$
$$(z\epsilon C, \phi(z)), (z\epsilon C, \psi(z)) \equiv (z\epsilon C, < \phi(z), \psi(Z) >),$$
$$(z\epsilon A \times B, \chi(z))^* \equiv (x\epsilon A, \lambda_{y\epsilon B}\chi(< x, y >)),$$
$$\varepsilon_{C,A} \equiv (y\epsilon C^A \times A, \varepsilon_{C,A}(\pi(y), \pi'(y))).$$

C(L) has a weak natural numbers object:

$$0 \equiv (x\epsilon 1, 0),$$
$$S \equiv (x\epsilon N, S(x)),$$
$$I_B \equiv (w\epsilon(B \times B^B) \times N, I(\pi(\pi(w)), \pi'(\pi(w)), \pi'(w)))$$

It is easy to make C into a functor λ-$Calc \longrightarrow \mathsf{Cart}_N$. Indeed, suppose $\phi : L \longrightarrow L'$ is a translation, define $C(\phi) : C(L) \longrightarrow C(L')$ as follows:

If A is an object of $C(L)$, that is , a type of $L, C(\phi)(A) = \phi(A)$ is the corresponding type of L', hence an object of $C(L')$.

If $f = (x\epsilon A, \varphi(x))$ is an arrow $A \longrightarrow B$ in $C(L)$, that is, $\varphi(x)$ is a term of type B in $L, C(\phi)(f) = (\phi(x)\epsilon\phi(A), \phi(\varphi(x)))$ is the correspondence arrow $\phi(A) \longrightarrow \phi(B)$ in $C(L')$.

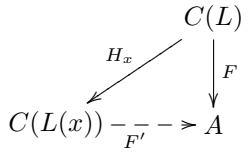
Proposition 7.3. *C is a functor from λ-Calc to Cart_N.*

Instead of adjoining an indeterminate arrow $x : 1 \longrightarrow A$ to the cartesian closed $C(L)$, one may adjoin a 'parameter' x of type A to the language L. To be precise , if L is a typed λ-*calculus* and x is a variable of type A, one may form another language $L(X)$ by adjoining the parameter x as follows:

$L(x)$ has exactly the same types as L and also the same terms , except that x is no longer counted as a variable. In other words, the closed terms of $L(x)$ are terms $\phi(x)$ in L which contain no free variables other than x. In the same spirit, $\overset{x}{=}$ in $L(x)$ means $\overset{X\cup\{x\}}{=}$ in L : just make sure that x is not in X.Some dictionaries define a 'parameter' as a 'varibale constant'. For us it is a variable kept constant.

Proposition 7.4. $C(L)[x] \cong C(L(x))$

Proof. We show that $C(L(x))$ has the universal property of $C(L)[x]$



The intermediate $x : 1 \longrightarrow A$ is defined by $(y\epsilon 1, x)$. H_x is C of the inclusion of L into $L(x)$, which may necessitate some relabelling of variable.Suppose $F : C(L) \longrightarrow A$ is any cartesian closed functor preserving the weak natural numbers object, and given arrow $b : 1 \longrightarrow F(A)$ in A, we claim that there is a unique such functor $F' : C(L(x)) \longrightarrow A$ such that $F'H_x = F$ and $F'x = b$.

Indeed, put $F'(B) = F(B)$ for each object B of $C(L)$, that is type in L. Suppose $f = (y\epsilon B, \varphi(x, y))$ is any arrow $B \longrightarrow C$ in $C(L(x))toA$,that is, $\varphi(x, y)$ is any term of type C in L with free variables $x\epsilon A$ and $y\epsilon B$. Define $F'(f) : F(B) \longrightarrow F(C)$ in A as follows:

First note that $\varphi(x,y) = \psi(y)^{\backprime}x$ holds, where $\psi(y)$ is $\lambda_{x\epsilon A}\psi(x,y)$. Thus

$$f = \varepsilon_{C,A} < g, x \bigcirc_B >$$

where $g = (y\epsilon B, \psi(y)) : B \longrightarrow C^A$ in $C(L)$. Now define

$$F'(f) = \varepsilon_{F(C),F(A)}\langle F(g), b \bigcirc_{F(B)}\rangle$$

.

That this definition has the right property is easily seen. Moreover it is clearly forced upon us, dince $F'(g) = F(g)$ and $F'(x) = b$. □

We are going to establish the main theorem:

Theorem 7.5. *The categories λ-Calc and Cart_N are equivalent, in fact $CL \cong id$ and $LC \cong id$.*

Proof. (i) Consider the natural transformation $\varepsilon : CL \longrightarrow id$ defined for each A in Cart_N by $\varepsilon(A) : CL(A) \longrightarrow A$ as follows:

An object of $CL(A)$ is a type of $L(A)$, that is, an object of A. Put $\varepsilon(A)(\mathrm{A}) = \mathrm{A}$.

An arrow $B \longrightarrow C$ in $CL(A)$ has the form $f = (y\epsilon B, \varphi(y))$, where $\varphi(y)\epsilon C$ in $L(A)$. Put $\varepsilon(A)(f) =$ the unique arrow $g : B \longrightarrow C$ such that $gy \stackrel{y}{=} \varphi(y)$, using functional completeness.

It is easily verified that $\varepsilon(A)$ is an arrow in Cart_N. Moreover, in view of functional completeness, it establishes a one-to-one correspondence between $Hom_C L(A)(B,C)$ and $Hom_A(B,C)$. Thus $\varepsilon(A)$ is an isomorphism.

(ii) Consider the natural transformation $\eta : id \longrightarrow LC$ defined for each L in λ-$Calc$ by $\eta(L) : L \longrightarrow LC(L)$ as follows:

$$\eta(L)(A) \equiv A$$

$$\eta(L)(\varepsilon(x_1, ..., x_n)) \equiv (z\epsilon 1, \varepsilon(x_1, ..., x_n)) in C(L(x_1, ..., x_n))$$

Note that we have identified $C(L)[x_1, ..., x_n]$ with $C(L(x_1, ..., x_n))$ as is justified by proposition 8.2. It is easily verified that $\eta(L)$ is an arrow in λ-$Calc$. To see that $\eta(L)$ is an isomorphism, construct its inverse, which sends $(z\epsilon 1, \phi(z))$ onto $\varphi(*)$ □

References

[1] Hugo Herbelin, *A λ-calculus structure isomorphic to sequent calculus structure*,Computer science logic: 8th workshop, CSL '94, Kazimierz, Poland, (September,1997).

[2] *http : //en.wikipedia.org/wiki/Curry − Howardcorrespondence.*

[3] J.Lambek and P.J.Scott, *Introduction to higer order categorical logic.*

[4] Michael Barr and Charles Wells, *Category Theory For Computing Science*,Third Edition(1992).

[5] Jean Gallier, *Constructive Logics.Part 1: A Tutorial on Proof Systems and Typed Λ-calculi*(May,1991).