The Curry-Howard correspondence for Propositional Logic

Edgar Lin

2022 February 28

1 Introduction

Intuitionistic logic attempts to formalize the the *BHK-interpretation* (named for Brouwer, Heyting, and Kolmogorov) of logical connectives. In particular, the BHK-interpretation of the logical connective $\rightarrow$ holds that a proof of $\varphi_1 \rightarrow \varphi_2$ is a program transforming any demonstration of φ_1 into one of φ_2. In order to demonstrate that a proof-system in intuitionistic logic properly formalizes the BHK-interpretation, then, we would like to show that any proof of $\varphi_1 \rightarrow \varphi_2$ does in fact correspond to a program in some formalism of computation that takes some representation of a proof of φ_1 to a proof of φ_2.

This correspondence is called the Curry-Howard correspondence. Here, we will explicate the Curry-Howard correspondence between the natural deduction proof-system for the implicational fragment of intuitionistic propositional logic, that is, the subset of intuitionistic propositional logic that only uses the logical connectives $\rightarrow$ and $\bot$, and the simply-typed lambda calculus, a formalism for representing computable functions.

The Curry-Howard isomorphism goes as follows: first, we interpret sentences in propositional logic as types of expressions in the simply-typed lambda calculus, in particular, we interpret sentences of the form $\varphi \rightarrow \psi$ as types of functions from expressions of the type corresponding to φ to expressions of the type corresponding to ψ. Then, a proof of a sentence is simply an expression in the simply-typed lambda calculus whose type is that sentence. The Curry-Howard correspondence, then, states that provability in intuitionistic propositional logic is equivalent to type-inhabitation in the

simply typed lambda-calculus by providing a correspondence between every natural deduction proof of a sentence in intuitionistic propositional logic and every term in the simply-typed lambda calculus such that the type of the lambda-term is the sentence to be proved. Further, this correspondence is computable by a structure-preserving transformation between the abstract syntax-trees of the proof and the program.

In the following sections, we will outline the implicational fragment of intuitionistic propositional logic (IPC($\rightarrow$)) along with the natural deduction proof system, before demonstrating the Curry-Howard isomorphism between the two systems and some of its uses and abuses. When it comes to demonstrating the computational properties of the correspondence, we will write our computations as Haskell programs instead of using formal systems of defining computations for the sake of readability.

2 The Natural Deduction Proof System for the IPC($\rightarrow$)

The syntactic rules for the implicational fragment of intuitionistic propositional formulae are much the same as that for classical logic. We start with a countably infinite set of propositional variables/atomic propositions PV. In the implicational fragment we only have one logical connective, $\rightarrow$, signifying implication. Then, our set of formulae/sentences Φ is the smallest set of finite strings of characters in containing all of the below:

- Every $A \in PV$,
- $(A \rightarrow B)$ for every $A, B \in \Phi$.

We note that in particular that we can't explicitly express $\bot$ in the implicational fragment. Thus, the implicational fragment is called a *positive* subset of the full intuitionistic logic. We will later show that this omission doesn't affect the set of (implicational) sentences we can prove.

We will assume the unique reading lemma. Then, we can equivalently define logical formulae as strings corresponding to the following abstract syntax tree (in Haskell):

```
module CurryHoward where
import qualified Data.Set as Set
```

```
import qualified Data.Map as Map
import qualified Data.List as List

type PV = String
data Formula = Atomic PV
  | Implies Formula Formula
  deriving (Show, Eq, Ord)
```

The conversion back to strings goes as follows:

```
formula :: Formula → String
formula (Atomic v) = v
formula (Implies p q) = "(" ++ formula p ++ "\\to " ++ formula q ++ ")"
```

A *theory* is a subset $\Gamma \subset \Phi$. We call a finite theory Γ a *context*. For a formula φ we wrote Γ, ϕ for $\Gamma \cup \{\varphi\}$.

```
type Theory = Set.Set Formula
```

For a context Γ and a formula φ we write $\Gamma \vdash \varphi$ to denote the fact that Γ proves φ. We call any φ such that $\vdash \varphi$ a theorem of IPC($\to$). A proof of $\Gamma \vdash \varphi$ is a (finite) tree whose nodes are labeled with context-formula pairs, which we will denote $\Delta \vdash \psi$, and that obey the following rules, and whose root is labeled $\Gamma \vdash \varphi$, such that the below hold:

- The leaf nodes are labeled $\Delta, \psi \vdash \psi$. These correspond to using an axiom in a proof.
- Non-leaf nodes have one or more children. The labels of parent nodes are inferred from those of their children by one of the following rules:
  - Introduction of $\to$ ($\to$ I): From $\Delta, \psi \vdash \pi$, infer $\Delta \vdash \psi \to \pi$.
  - Elimination of $\to$ ($\to$ E): From $\Delta \vdash \psi \to \pi$ and $\Delta \vdash \psi$, infer $\Delta \vdash \pi$.

We can write a natural deduction tree and proof-checking rules in Haskell as follows:

```
data Proof = Axiom (Theory, Formula)
  | Introduction (Theory, Formula) Proof
  | Elimination (Theory, Formula) Proof Proof
```

```
  deriving (Show, Eq)
label :: Proof → (Theory, Formula)
label (Axiom a) = a
label (Introduction a _) = a
label (Elimination a _ _) = a
valid :: Proof → Bool
valid (Axiom (g, p)) = Set.member p g
valid (Introduction (g, Implies p q) c) =
  (fst $ label c) ≡ Set.insert p g ∧ (snd $ label c) ≡ q
valid (Introduction _ _) = False
valid (Elimination (g, p) major minor) =
  (fst $ label major) ≡ g
   ∧ (fst $ label minor) ≡ g
   ∧ (snd $ label major) ≡ Implies (snd $ label minor) p
```

We note that we usually use the following notation to write nodes of natural deduction trees:

$$\frac{Children}{Label},$$

2.1 Natural Deduction & BHK

We can interpret the rules of natural deduction in BHK as follows. First, we take $\{\varphi_1, \varphi_2, \ldots, \varphi_n\} \vdash \psi$ as "Given (black-box) demonstrations for each φ_i, there is some construction built out of the φ_i that is a demonstration of ψ." Then, we can interpret the $\to$I rule, $\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi}$ as follows: "We have that given demonstrations for the contents of Γ and for ϕ we can build out of those demonstrations a demonstration of ψ. Then, using the demonstrations for the contents of Γ, we can write that demonstration of ψ with the demonstrations for ϕ erased and then write a program that takes as input a new demonstration of ϕ and inserts it into the blanks where the old demonstrations for ϕ were. This program takes as input a demonstration of ϕ and produces a demonstration of ψ, so it is a proof of $\phi \to \psi$." Finally, the $\to$E rule can be interpreted as applying the program that is the proof of the major premise to the proof of the minor premise to get a proof of the conclusion.

2.2 Natural Deduction & Hilbert-Style Proofs

Let's compare natural deduction for intuitionistic logic to the Hilbert-style proof system for classical logic where we had only one inference rule, modus ponens, and instead had three axiom schemes (we assume that $\to$ associates to the right for brevity):

P_1: $\phi \to (\psi \to \phi)$.

P_2: $(\phi \to (\psi \to \chi)) \to (\phi \to \psi) \to (\phi \to \chi)$.

P_3: $((\phi \to \bot) \to \bot) \to \phi$.

The $\to$E rule clearly corresponds to modus ponens.

What does $\to$I correspond to? We note that P_1 and P_2 in the Hilbert proof system are sufficient to prove the deduction theorem:

Theorem 1 (Deduction Theorem)**.** *In the Hilbert proof system with only axioms P_1 and P_2, if $\Gamma, \phi \vdash \psi$, then, $\Gamma \vdash \phi \to \psi$.*

Further, we can show that every member of the axiom schemes P_1 and P_2 are theorems of IPC($\to$) in natural deduction:

Theorem 2. *For any triplet of propositions (ϕ, ψ, χ), denote their corresponding axioms in the axiom schemes P_1 and P_2 as $P_{1,\phi,\psi}$ and $P_{2,\phi,\psi,\chi}$ respectively. Then, for all $\phi, \psi, \chi \in \Phi$, we have that in IPC($\to$), $\vdash P_{1,(\phi,\psi)}$ and $\vdash P_{2,(\phi,\psi,\chi)}$.*

Proof. For any ϕ, ψ, the following is a natural deduction proof of $P_{1,\phi,\psi}$:

$$\dfrac{\dfrac{\phi, \psi \vdash \phi}{\phi \vdash (\psi \to \phi)}}{\vdash (\phi \to (\psi \to \phi))}.$$

The proof of $P2$ is omitted for brevity. □

3 Simply-Typed Lambda Calculus

We remember that in the BHK-interpretation a proof $\varphi \to \psi$ should correspond to a program that takes a proof of φ and returns a proof of ψ. If we are to identify proofs with programs, then a proof of $\varphi \to \psi$ would then be

identified with a program that takes a program and returns another program. It would then be desirable to have a formal model of computation where the basic domain of programs are other programs. This model of computation will be called the *simply-typed lambda calculus*.

Here, we will present the Church-system of simply-typed lambda calculus. We will note that there is also a Curry-system of simply-typed lambda calculus, that is mostly equivalent.

3.1 Simple Types and Pre-Terms

In the simply-typed lambda calculus, we have two sorts of objects: types and terms. Both types and terms will be using *syntactic* rules; meanwhile, terms will be assigned at most one type by the *typability* relation; we can think of the subset of terms that can be assigned types as the set of terms that are *semantically* well-defined. A type in the simply-typed lambda calculus will be a string following the same syntactic rules as propositions in IPC($\rightarrow$). Thus we can immediately identify the set of simple types with Φ.[1] In the context of the simply-typed lambda calculus, members of PV are called *type variables*. If σ and τ are types and α is a type variable, we write $\sigma[\alpha := \tau]$ to denote the type resulting from substitution of τ for every occurence of α in σ.

Syntactically, programs in the lambda-calculus are first written in terms of the rules for forming *pre-terms*. We begin with a countably infinite alphabet of variables V. Then, the set Λ^-_Φ of pre-terms is the smallest set of strings containing V and closed under the following rules:

- If $x \in V$, $\phi \in \Phi$, and $M \in \Lambda^-_\Phi$, then, $(\lambda x.\phi\ M) \in \Lambda^-_\Phi$. This rule is called *abstraction* and can be roughly interpreted as defining a function $f\colon \phi \rightarrow ?$ that can be written as $f(x) = M$.
- If $M, N \in \Lambda^-_\Phi$, then, $(M\ N) \in \Lambda^-_\Phi$. This rule is called *application*; if M evaluates to some function f this should represent $f(N)$.

[1] The types in the simply-typed lambda calculus are called "simple" as they lack (a) parameters or (b) recursion. For example, the following *List* type definition in Haskell would not be a simple type:

List a = Cons a (List a) | Empty

That is, Λ_Φ^- corresponds to the set of strings representing the following Haskell data structure:

type *V* = *String*
data *Term* = *Variable V*
| *Abstraction V Formula Term*
| *Application Term Term*

For a pre-term $M \in \Lambda_\Phi^-$ in the lambda calculus, we call the set of variables x appearing in M outside of an abstraction term $(\lambda x.\phi\ M)$ the set of *free variables* $FV(M)$. More formally, the set of free variables is defined inductively such that

- $FV(x) = \{x\}$,
- $FV(\lambda x.\phi\ M) = FV(M) \setminus \{x\}$,
- $FV(M\ N) = FV(M) \cup FV(N)$.

3.2 Typability

We can see the free variables of terms in the lambda-calculus as referring to black-box programs of some type provided by an outside context, and computation in the lambda-calculus as manipulations of those programs.

We would like to see be able to check that given the black-box programs supplied by our context, the manipulations on those programs we define in a lambda-term make semantic sense. In particular, when we our context supplies a program f with type $\phi \to \psi$, we mean that (fx) (which we recall is the lambda calculus expression representing $f(x)$) only has a well defined meaning when x has type ϕ, so we want to make sure the computation our term specifies only ever applies f to terms of type ϕ.

Thus: we call a finite $\Gamma \subset V \times \Phi$ such that Γ defines a function from its *domain* $\{x \in V \mid \exists\phi((x,\phi) \in \Gamma)\}$ to Φ a *context*. We denote the range of the function Γ defines as $|\Gamma|$.

type *Context′* = *Map.Map V Formula*

Denote by C the set of such contexts Γ. We denote the pairs $(x,\phi) \in \Gamma$ as $x : \phi$, which we interpret as Γ assigning the type ϕ to the variable x. We

note that we use the same shorthands for denoting contexts in intuitionistic logic: eg, $\Gamma, x : \phi$ denotes $\Gamma \cup \{x : \phi\}$.

Then, we can define a three-way relation between types $\Gamma \in C$, pre-terms $M \in \Lambda^-_\Phi$, and types $\phi \in \Phi$ called *typability*, which we denote as $\Gamma \vdash M : \phi$.

We define typability inductively and case-wise as follows:

- If M is a free variable $x \in V$, then, $\Gamma \vdash M : \phi$ if and only if $x : \phi \in \Gamma$.
- If M is equal to an application term (PQ), then, $\Gamma \vdash M : \phi$ if and only if there exists some ψ such that $\Gamma \vdash P : \psi \to \phi$ and $\Gamma \vdash Q : \psi$.
- If M is equal to an abstraction term $\lambda x : \psi\ N$, then, $\Gamma \vdash P : \phi$ if and only if there exists some τ such that $\phi = \psi \to \tau$ and $\Gamma, x : \psi \vdash N : \tau$. In this term we treat the abstraction term as supplying the context of N with an additional variable x of type ψ, and $\lambda x : \psi\ N$ as having the type of a function with domain ψ and the codomain being whatever type N has when evaluated after being supplied an x of type ψ.

We call pre-terms M such that there exists some Γ and some ϕ such that $\Gamma \vdash M : \phi$ *typable*. We note that if a pre-term M has a type under Γ, that type is unique.

Finally, we note the substitution lemma:

Lemma 3 (Substitution)**.** *If* $\Gamma \vdash M : \phi$*, then,* $\Gamma[\alpha := \tau] \vdash M[\alpha = \tau] : \phi[\alpha := \tau]$.

3.3 Substitution and α-equivalence

The basic model of lambda calculus is computation by executing symbolic substitution rules. We will want to define two definitions of equivalence in the lambda-calculus. First, $\cong_\alpha$ will denote the fact that two strings (pre-terms) represent the same program, ie, the same substitution rules. Then, $\cong_\beta$ will denote the fact that two programs represent the same *value*, ie, they will eventually evaluate to the same thing.

For a pre-term M we write $M[x := N]$ to denote the substitution of N for x in M. In doing so, we replace every *free* instance of x in M by N, while preserving the set of free variables of N. Thus substitution is defined inductively as follows:

- $x[x := N] = N$

- $y[x := N] = y$ for $y \neq x$.
- $(P\ Q)[x := N] = (P[x := N]\ Q[x := N])$.
- $(\lambda x.\phi\ P)[x := N] = (\lambda x.\phi\ P)$: instances of x in P are not free, so we don't replace them.
- $(\lambda y.\phi\ P)[x := N] = (\lambda y.\phi\ P[x := N])$ when $y \neq x$ and $y \notin FV(N)$: in this case the instances of x in P are still free, so should be replaced. We note that we run into a bit of difficulty when y is a free variable of N: we want substitution to preserve the semantic meaning of N, wherein y is a free variable referring to something in the outside context.

  Substituting N directly into P would cause y to become bound by λy, changing the meaning of N as the instances of y in N would now refer to the argument variable y of the lambda term.

  The solution is to stipulate that the names of argument variables used in a function definition shouldn't matter: ie, $f(x) = g(x)$ should define the same function as $f(y) = g(y)$. Thus, we will rename the y variable in the original abstraction term to something that doesn't conflict with N before substituting N in. This leads us to our last rule:
- $(\lambda y.\phi\ P)[x := N] = (\lambda z.\phi\ P[y := z][x := N])$ for some choice of $z \notin FV(P) \cup FV(N)$ when $y \neq x$ and $y \in FV(n)$.

We want to expand on a bit more on the names of argument variables not mattering. We will eventually want to define $(\lambda x.\phi\ N)\ M$ as representing the substitution rule where we substitute all occurences of x in N with M. Its clear that the variable x here is strictly temporary, and thus $(\lambda y.\phi\ N[x := y])M$ would represent the same substitution operation as both x and y will be replaced with M anyway.

Thus we define the $\cong_\alpha$ as the smallest equivalence relation on Λ_Φ^- such that $(\lambda x : \phi\ N) = (\lambda y : \phi\ N[x := y])$ and that is preserved under application and abstraction.

We can thus define the set of *terms* Λ_Φ in the simply-typed lambda calculus by $\Lambda_\Phi = \Lambda_\Phi^- / \cong_\alpha$. Thus, while Λ_Φ^- represents the set of distinct representations of programs, Λ_Φ can be seen as representing the set of distinct programs themselves.

It should be fairly easy to see that if $P \cong_\alpha Q$ then $P[x := N] \cong_\alpha Q[x := N]$, and that if $P \cong_\alpha Q$, then, $\Gamma \vdash P : \phi$ if and only if $\Gamma \vdash Q : \phi$. It follows that substitution and typability are also defined for terms Λ_Φ.

We proceed to define what it means to evaluate a program in the lambda calculus.

3.4 β-reduction

We mentioned in the previous section that the abstraction term represents a symbolic substitution rule that can be evaluated when applied to another term by an application term. That is, $(\lambda x : \phi\ N)\ M$ evaluates to $N[x := M]$. We represent evaluation by the relation $\rightarrow_\beta$, which is defined to be the smallest relation such that $((\lambda x : \phi\ N)\ M) \rightarrow_\beta N[x := M]$ and $\rightarrow_\beta$ is preserved under application and abstraction.

That is, $P \rightarrow_\beta Q$ if we can find somewhere in P some application term applying an abstraction term to another term and replacing that application term with its evaluated form yields Q. We note that if P contains multiple such application terms, we can choose different application terms to evaluate in each step to yield multiple Q_i such that $P \rightarrow_\beta Q_i$.

We note that if $\Gamma \vdash P : \phi$ and $P \rightarrow_\beta Q$, then, $\Gamma \vdash Q : \phi$. We note that the converse is not true, as P may be untypable, and an untypable term may reduce to a typable one.

We define the multi-step β-reduction relation $\twoheadrightarrow_\beta$ to be the transitive-reflexive closure of $\rightarrow_\beta$, that is, $P \twoheadrightarrow_\beta Q$ if there is a chain of zero or more evaluations that reduces P to Q.

We call the transitive-reflexive-symmetric closure of $\rightarrow_\beta$ β-equivalence, or $\cong_\beta$.

Finally, we call a term N that does not reduce to any other term M β-normal. We can see β-normal forms as the results of a halted chain of computations in the lambda-calculus.

We will now state a few more results without proof about β-reduction and typability before moving on to prove the Curry-Howard Isomorphism.

3.4.1 Church-Rosser Theorem

We would like to know thar if $P \cong_\beta Q$ then there is some evaluation path such that P and Q eventually evaluate to the same term.

Further, we would like to know that the β-normal form that results from an evaluation path of some P is unique, that is, there do not exist $Q \neq R$ such that both Q and R are β-normal and both $P \twoheadrightarrow_\beta Q$ and $P \twoheadrightarrow_\beta R$.

These properties are guaranteed by the Church-Rosser Theorem.

Theorem 4 (Church-Rosser)**.** *Let $P \twoheadrightarrow_\beta Q$ and $P \twoheadrightarrow_\beta R$. Then, there exists some S such that $Q \twoheadrightarrow_\beta S$ and $R \twoheadrightarrow_\beta S$.*

Two corollaries immediately follow.

Corollary 5. *For all $P \in \Lambda_\Phi$, there exists at most one β-normal $Q \in \Lambda_\Phi$ such that $P \cong_\beta Q$.*

Corollary 6. *If both $P \in \Lambda_\Phi$ and $Q \in \Lambda_\Phi$ are typable under Γ, and $P \cong_\beta Q$, then P and Q have the same type under Γ.*

3.5 Weak-Normalization

We now present one way in which a term being typable is a guarantee of it being semantically well defined.

Theorem 7 (Weak Normalization)**.** *If P is typable, then there exists some β-normal N such that $P \twoheadrightarrow_\beta N$.*

This theorem implies that typability implies that the computation expressed by the lambda term halts.

It follows that the *typable* subset of the simply-typed lambda calculus is not Turing complete.

Remark. *We will note that the weak-normalization property is not true of terms in general. In particular, if we ignore the typability restriction we can define the fixed-point or Y combinator that has the property that $f\ (Y\ f) \cong_\beta Y\ f$. This allows the us to implement full recursion, including terms that never terminate.*

We note that while there is no typable version of Y, sometimes $Y\ f$ can β-reduce to a typable term.

4 The Curry-Howard Correspondence

Now that we've provided the basic properties of both natural deduction and the simply-typed lambda-calculus, we can present the correspondence.

Theorem 8. *Let $\phi \in \Phi$. Then, a context $\Gamma \vdash \phi$ in natural deduction if and only if $\Gamma = |\Gamma'|$ where Γ' is a context in the simply-typed lambda calculus and there exists some lambda term $M \in \Lambda_\Phi$ such that $\Gamma' \vdash M : \phi$.*

Proof. First we prove that $\Gamma' \vdash M : \phi$ implies that $|\Gamma'| \vdash \phi$. We claim there is a surjection χ from $\{(\Gamma, M) \in C \times \Lambda_\Phi \mid \text{M is typable under } \Gamma\}$ to proofs in natural deduction such that if $\Gamma' \vdash M : \phi$, then $\chi(\Gamma', M)$ is a proof of $|\Gamma'| \vdash \phi$. We define $\chi(\Gamma', M)$ inductively as follows:

```
range :: Context' → Context
range = Set.fromList ∘ Map.elems
termToProof :: Context' → Term → Maybe Proof
```

First, when $M = x$, we have that if $\Gamma' \vdash x : \phi$ for some ϕ, then $x : \phi \in \Gamma'$, so $\phi \in |\Gamma'|$. Then, we can define $\chi(\Gamma', M) = (|\Gamma'| \vdash \phi)$, and $\chi(\Gamma', M)$ is a proper proof of $|\Gamma'| \vdash \phi$ under the axiom rule. This corresponds to the following code in Haskell:

```
termToProof c (Variable v) = fmap (Axiom ∘ (range c, )) $ Map.lookup c v
```

Secondly, when $M = (P\ Q)$, we get that $\Gamma' \vdash M : \phi$ only if there exists some ψ such that $\Gamma' \vdash P : \psi \to \phi$ and $\Gamma' \vdash P : \psi$.

Then,

$$\chi(\Gamma', M) = \frac{\chi(\Gamma', P)\ \chi(\Gamma', Q)}{|\Gamma'| \vdash \phi}$$

is a proof of $|\Gamma'| \vdash \phi$ using the $\to$E rule as $\chi(\Gamma', P)$ is a proof of $\psi \to \phi$ and $\chi(\Gamma', Q)$ is a proof of ψ. Or, in Haskell,

```
termToProof c (Application p q) =
  case (termToProof c p, termToProof c q) of
    (Just proofP, Just proofQ) →
      case (label proofP, label proofQ) of
        ((_, Implies psi phi), (_, psi')) | psi ≡ psi' →
          Just Elimination (range c, phi) proofP proofQ
```

```
        _ → Nothing
    _ → Nothing
```

Finally, when $M = (\lambda x : \phi\ N)$, we get that if M is typable under Γ' then for some τ, $\Gamma' \vdash M : \phi \to \tau$ and $\Gamma', x : \phi \vdash N : \tau$.

Then,

$$\chi(\Gamma', M) = \frac{\chi(\Gamma' \cup \{x : \phi\}, N)}{|\Gamma'| \vdash \phi \to \tau}$$

is a valid natural deduction proof of $|\Gamma'| \vdash \phi \to \tau$ under the $\to$I rule as $\chi(\Gamma' \cup \{x : \phi\})$ is a valid proof of $|\Gamma'|, \phi \vdash \tau$. Or, in Haskell,

```
termToProof c (Abstraction x phi n) =
  fmap
    (λproofN →
       Introduction (range c, Implies phi (snd $ label proofN)) proofN
    )
   $ termToProof (Map.insert x phi c) n
```

Then, if we show that χ is a surjection, we prove the equivalence.

We note however that for any proof of $\Gamma \vdash \phi$ we can construct a Γ', M such that $\chi(\Gamma', M)$ is a proof of $\Gamma \vdash \phi$ as follows. First, construct Γ' from $\Gamma = \{\psi_1, \psi_2, \ldots, \psi_n\}$ by choosing $\{x_1, x_2, \ldots, x_n\} \in V$ and letting $\Gamma' = \{x_1 : \psi_1, x_2 : \psi_2, \ldots, x_n : \psi_n\}$.

Then, for Γ' such that $|\Gamma'| = \Gamma$ and a proof of $\Gamma \vdash \phi$ we construct the corresponding lambda term inductively as follows:

If the proof is an axiom $\Gamma, \phi\colon \phi$, then we have that there exists some i such that $\psi_i \in \Gamma$ so $\Gamma' \vdash x : \phi$.

If the proof is a $\to$E rule, with P and Q being the proofs of $\psi \to \phi$ and ψ respectively, we simply apply the lambda term corresponding to Γ' and P to the one corresponding to Γ' and Q.

If the proof is a $\to$I rule, with $\phi = \sigma \to \tau$ and N being a proof that $\Gamma, \sigma \vdash \tau$, then, choose some $x \in V$ such that x is not in the dommain of Γ'. Then the proof corresponds to the abstraction term $(\lambda x : \sigma\ N')$ where N' is the term corresponding to N under the augmented context $\Gamma' \cup \{x : \sigma\}$.

The resulting term corresponds to the original proof under the map χ, thus χ is surjective. □

4.1 Application: Consistency

We can use the correspondence to prove the consistency of IPC($\to$) in the case of an empty context. Here, consistency is defined to mean that there exists some formula ϕ such that $\nvdash \phi$.

Theorem 9. *IPC($\to$) is consistent.*

Proof. Suppose not. Then, let ϕ be a propositional variable: we must have that $\vdash \phi$.

But that means that there must be some term M such that $\vdash M : \phi$. Since M is typable, it must be reducible to a β-normal form M' that has the same type. Consider the root of the syntax tree of M': M' cannot be a variable x, since then $x : \phi \in \{\}$ by the typability rule.

If M' is an application term, then we note that its left child cannot be an abstraction term, since otherwise M' wouldn't be β-normal. We further note that since subterms of β-normal terms are themselves β-normal, if M' is an application term such that the depth of the left-most leaf is h, either $h = 1$ in which case $M' = (x\ Q)$ for $x \in V$, or $M' = (P\ Q)$ for an application term with left-height $h - 1$. But, if $M' = (x\ Q)$ then $\vdash M' : \phi$ only if $\vdash x : \psi \to \phi$ for some ψ, but we know that is not true already. However, if for all application terms with left-height h we have that $\nvdash P : \phi$ for any ϕ, then we get that $\nvdash (P\ Q) : \phi$ for any ϕ. So M' is not an application term.

Then, M' is an abstraction term. But abstraction terms are only typable by types of the form $\sigma \to \tau$, so if ϕ is a propositional variable $\nvdash M' : \phi$, so $\nvdash \phi$.

So, IPC($\to$) is consistent. □

4.2 Application: Untypability of the Y combinator

While the untypability of the Y combinator follows from the weak-normalization property and thus the Turing-incompleteness of the typable subset of the lambda-calculus, it is also provable using the Curry-Howard isomorphism.

In particular, we show Y is a proof of inconsistency.

Theorem 10. *Let σ be a propositional variable. Let Y be a term with no free variables such that for any $f : \sigma \to \sigma$, $f(Y\ f) \cong_\beta Y\ f$. Then, Y is not typable.*

Proof. Since Y has no free variables it is typable only if there is some ϕ such that $\vdash Y : \phi$. If Y is typable, then, $(Y\ f)$ is, and $(f\ (Y\ f))$ is, and, since they are β-equivalent, those two types are the same. But we note that $(Y\ f)$ is typable only if ϕ has the form $(\sigma \to \sigma) \to \tau$ for some τ and $(f\ (Y\ f))$ is typable only if that τ is equal to σ.

So, if Y were typable then $\vdash Y : (\sigma \to \sigma) \to \sigma$.

By the substitution lemma, we have that for any type ϕ we have that $\vdash Y[\sigma := \phi] : (\phi \to \phi) \to \phi$. But that means that for any ϕ we have that in natural deduction $\vdash (\phi \to \phi) \to \phi$. Then, we since for any ϕ we have that $\vdash \phi \to \phi$, we have that $\vdash \phi$ for any ϕ.

But that means that natural deduction is inconsistent, which we know is not true. So Y is untypable. □

References

[1] Morten Heine B. Sørensen & Paweł Urzyczyn, *Lectures on the Curry-Howard Isomorphisms.*