Relationship between Curry-Howard-correspondence and semantics

Gyesik Lee

gslee@ropas.snu.ac.kr

ROSAEC Center

Seoul National University

April 10, 2009

Contents

- Curry-Howard correspondence
- Normalization and cut-elimination
- Program extraction from proofs
- Kripke semantics and forcing
- Consequences and future work

Why does logic matter?

Two seeming irrelevant areas have proven to be closely connected.

Why does logic matter?

> *Two seeming irrelevant areas have proven to be closely connected.*

Indeed, logic matters more to computer science than to mathematics although logic emerged from mathematics.

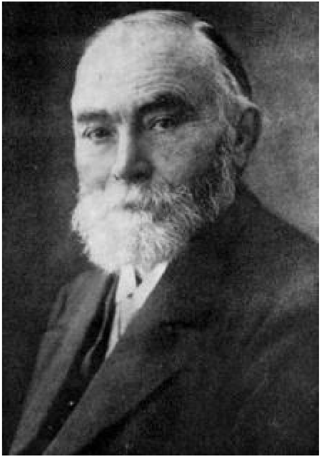
Curry-Howard correspondence

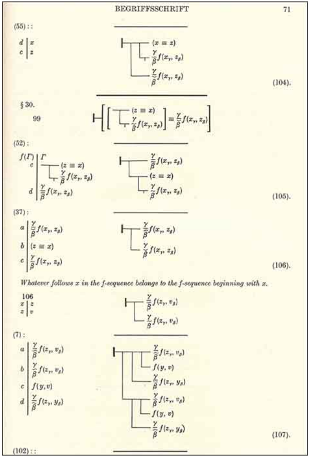
- Relationship between models of computation (computer programs) and proof systems.
- A proof is a program, the formula it proves is a type for the program.
- An underlying principle connecting typed λ-calculus and proof theory.
- Programs : (inputs : assumptions) $\Rightarrow$ (outputs : theorems)

Origin

- (1934) Haskell B. Curry: the types of the combinators could be seen as axiom-schemes for intuitionistic implicational logic.
- (1969) William A. Howard: the natural deduction system can be directly interpreted in the simply typed λ-calculus.
- (1990) Timothy Griffin: extension of the correspondence to classical logic
- (1992) Michel Parigot: $\lambda\mu$-calculus is invented in order to be able to describe expressions corresponding theorems in classical logic.

Frege's Begriffsschrift (concept notation), 1879





BEGRIFFSSCHRIFT 71

(55) : :

(104).

§ 30.

99

(52) :

(105).

(37) :

(106).

Whatever follows x in the f-sequence belongs to the f-sequence beginning with x.

106

(7) :

(107).

(102) : :

Constructive understanding of $A \rightarrow B$

- $(A \rightarrow B)$ vs. $(\neg A \vee B)$

Constructive understanding of $A \longrightarrow B$

- $(A \longrightarrow B)$ vs. $(\neg A \vee B)$
- How about
$$(A \longrightarrow A) \quad \text{vs.} \quad (\neg A \vee A)$$
where A is undecidable or not decided yet?

Constructive understanding of $A \to B$ (Cont.)

- A proof of $A \to B$ is a construction that converts a proof of A into a proof of B.

Constructive understanding of $A \to B$ (Cont.)

- A proof of $A \to B$ is a construction that converts a proof of A into a proof of B.

- A proof of $A \to B$ is a function (program) that converts a proof of A into a proof of B.

Simply typed lambda calculus ($\lambda\!\to$)

- Types:
  - $P, Q \ldots$ are types.
  - With A, B types, $A \to B$ is a type.
- Well typed terms: $\Gamma = x_1 : A_1, \ldots, x_n : A_n$

$$\frac{}{\Gamma \vdash x_i : A_i}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A . t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}$$

- β-reduction of redexes:

$$(\lambda x^A . t)\, u \to_\beta t[x := u]$$

IPC($\to$) vs. $\lambda^{\to}$

Let $\Gamma := A_1, ..., A_n$

$$\frac{}{\Gamma \vdash A_i}\ (Ax)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (Imp) \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (Cut)$$

IPC($\to$) vs. $\lambda^{\to}$

Let $\Gamma := A_1, ..., A_n$

$$\frac{}{\Gamma \vdash A_i}\ (Ax)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (Imp) \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (Cut)$$

- $\Gamma \vdash A$ if and only if $\exists t\, (\Gamma \vdash t : A)$.

Curry-Howard correspondence

$\lambda\rightarrow$	IPC($\rightarrow$)
term variable	assumption
term	construction (proof)
type variable	propositional variable
type	formula
type constructor	connective
inhabitation	provability
typable term	construction for a proposition
redex	cut-rule
reduction	normalization
value	normal construction

Curry-Howard correspondence (Cont.)

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

Calculus of Constructions

- Dependent types
- Polymorphism
- Adaption of Martin-Löf's constructive meta-theory to a concrete type system
- Reflection of the Curry-Howard correspondence

CIC: extension with inductive types

- CC with various notions of type definitions provided in conventional programming languages.
- Similar to the recursive type definitions used in most functional programming languages.
- More precise and expressive by combination of recursive types and dependent products
- Each inductive type corresponds to a computation structure, based on pattern matching and recursion.
- The basis theory for the proof assistant Coq.

Normalization vs. cut-elimination

- Cut elimination corresponds to normalization and vice versa.
- Why is it important to have these properties?
  - Subformula property $\Rightarrow$ the possibility of carrying out proof search based on resolution.
  - Decidability of provability (in many propositional logic systems) $\Rightarrow$ Propositional logic is decidable $\Rightarrow$ decidability of inhabitation $\Rightarrow$ existence decidability of a program
  - Consistency of a system.

Program extraction from proofs

- How to prove the correctness of a program?
- Curry-Howard correspondence provides tools to produce certified programs.
- In case of ($\lambda\rightarrow$), we first write a formula within a appropriate language that can describe the specification of a program, then prove the formula withing a theorem prover like Coq, then extract the program.
- Not necessarily efficient programs, but program extraction has become an important research area, or developing programming languages with polymorphic and dependent types such as Coq, Agda (extension of Haskell with dependent types), Epigram, etc.

Program extraction from proofs (Cont.)

- How to prove the cut-elimination?
- How to write a program producing normal proof terms?
- Both are main topics when establishing a theory or a type system.

From semantics to rules

Given a proof t for a sentence A, we would like to construct a cut-free proof t' with the same result A.

From semantics to rules

Given a proof t for a sentence A, we would like to construct a cut-free proof t' with the same result A.

$$t \quad \Longrightarrow \quad [\![t]\!] \quad \Longrightarrow \quad t'$$

such that $t \cong t'$.

$$\Gamma \vdash A \quad \stackrel{S}{\Longrightarrow} \quad \Gamma \Vdash A \quad \stackrel{C}{\Longrightarrow} \quad \Gamma \vdash A$$

S : soundness, C : cut-free completeness

From semantics to rules

Given a proof t for a sentence A, we would like to construct a cut-free proof t' with the same result A.

$$t \quad \Longrightarrow \quad [\![t]\!] \quad \Longrightarrow \quad t'$$

such that $t \cong t'$.

$$\Gamma \vdash A \quad \stackrel{S}{\Longrightarrow} \quad \Gamma \Vdash A \quad \stackrel{C}{\Longrightarrow} \quad \Gamma \vdash A$$

S : soundness, C : cut-free completeness

- Everything is formalized in a theorem prover such as Coq.
- Combination of S and C leads to a automated cut-elimination (normalization) program.

From semantics to rules (Cont.)

- U. Berger and H. Schwichtenberg, *An inverse of the evaluation functional for typed λ-calculus*. (1991)
- C. Coquand, *From semantics to rules: A machine assisted analysis*. (1993)
- H. Herbelin and G. Lee, *Forcing-based cut-elimination for Gentzen-style intuitionistic sequent calculus*. (2009)
- D. Ilik, G. Lee, and H. Herbelin, *Kripke models for classical logic*. (2009)

Kripke semantics

- A formal semantics for classical logic systems created in the late 1950s and early 1960s by Saul Kripke
- First made for modal logic, and later adapted to intuitionistic logic and other non-classical systems.
- Extension to classical systems using double negation or modifying Krivine's realization method.
- In classical systems, the Curry-Howard correspondence and Kripke semantics can also be used to express the duality between the two evaluation strategies known as call-by-name and call-by-value. (Cf. Curien and Herbelin, 2000)

Comment: classification of formal methods

The whole process follows the following classification of formal methods:

- Formal specification:

  - description of what systems should do

  - based on a formal language syntax

- Formal verification:

  - proving or disproving the correctness of intended algorithms

- Automated theorem prover:

  - proving of mathematical theorems by a computer program

Consequences and future work

- Very simple proof of cut-elimination (normalization)
- Direct relationship between syntax and semantics
- Mechanization of proofs $\Rightarrow$ contribution for a more easy formalization technique.

Consequences and future work

- Very simple proof of cut-elimination (normalization)
- Direct relationship between syntax and semantics
- Mechanization of proofs $\Rightarrow$ contribution for a more easy formalization technique.
- Extension of forcing-based cut-elimination to more powerful theories.
- More efficient program extraction is necessary.