The Curry-Howard Isomorphism

CSCI 2210

2023-11-27

Logic in Pure Lambda Calculus

We can encode logic and arithmetic in pure λ-calculus using *Church encodings*.

However, this is not very pleasant to work with.

Pure λ-calculus is equivalent in computational power to Turing machines.

However, this means that the *normalization problem* is not decidable.

The λ-calculus was intended as a tool to study mathematical logic.

But nonterminating β-reductions make it unsuitable for this purpose.

Strength Through Weakness

To use λ-calculus for mathematical logic people sought to *restrict* the set of admissible terms to those that can be interpreted into sets and functions.

This excludes terms like:

- $\lambda x \,.\, x\, x$, because no mathematical function is an element of its own domain,
- $(\lambda x \,.\, x\, x)\,(\lambda x \,.\, x\, x)$, which has no normal form.

One way to do this is to begin with a set of primitive terms and inductively add terms of certain forms that we know preserve desired properties.

This is the idea behind **simply-typed λ-calculus** (STLC).

Types of STLC

We start with an arbitrary finite set of **base types** ι.

A **type** of the simply-typed λ-calculus is given by the following inductive definition:

base type: If $\mathrm{A} \in \iota$ then A is a type.

product types: If A and B are types then $\mathrm{A} \times \mathrm{B}$ is a type.

function types: If A and B are types then $\mathrm{A} \to \mathrm{B}$ is a type.

unit type: Unconditionally, 1 is a type.

By convention:

- $\times$ binds more tightly than $\to$, so "$\mathrm{A} \times \mathrm{B} \to \mathrm{C}$" means $(\mathrm{A} \times \mathrm{B}) \to \mathrm{C}$,
- $\to$ associates to the right, so "$\mathrm{A} \to \mathrm{B} \to \mathrm{C}$" means $\mathrm{A} \to (\mathrm{B} \to \mathrm{C})$.

The set of types is called "Ty".

Terms of STLC

We start with a countably infinite set of **variables** V.

A **term** of the simply-typed λ-calculus is given by the following inductive definition:

variable: If $x \in \mathrm{V}$ then x is a term.

application: If M and N are terms then $\mathrm{M}\,\mathrm{N}$ is a term.

abstraction: If x is a variable, A is a type, and M is a term then $\lambda\, x{:}\mathrm{A}\;.\;\mathrm{M}$ is a term.

pairing: If M and N are terms then $\langle \mathrm{M}\;,\;\mathrm{N} \rangle$ is a term.

projection: If M is a term then $\pi_0\,\mathrm{M}$ and $\pi_1\,\mathrm{M}$ are terms.

it: Unconditionally, $\star$ is a term.

The set of terms is called "Tm".

Type Assignment

In STLC we admit only those terms that are **well-typed**.

These are terms to which we can give a **type assignment**.

The notation "$\mathrm{M} : \mathrm{A}$" means "term M has the type A".

The type of a term depends on the types of its subterms, and ultimately on the types of its variables.

A **typing context** is a *partial function* from variables to types $\Gamma : \mathrm{V} \rightharpoonup \mathrm{Ty}$ giving type assignments for variables.

A context is typically written as a sequence, "$x_0 : \mathrm{A}_0, x_1 : \mathrm{A}_1, \cdots, x_n : \mathrm{A}_n$".

Typing Judgements

A **typing judgement** is a *type assignment* for a term in a *context* that includes all of its free variables.

It is typically written using **sequent** notation:

$$\overbrace{\Gamma}^{\text{typing context}} \vdash \underbrace{\mathrm{M} : \mathrm{A}}_{\text{type assignment}}$$

For example, the typing judgement

$$\Gamma, x : \mathrm{A} \vdash x : \mathrm{A}$$

means, "in a context where the variable x has type A, the term consisting the variable x has type A".

Typing Rules

The typing judgements of simply-typed λ-calculus are inductively generated by:

Variables:

$$\frac{}{\Gamma, x : \mathrm{A} \vdash x : \mathrm{A}} \ \mathrm{var}$$

Function Types:

$$\frac{\Gamma, x : \mathrm{A} \vdash \mathrm{M} : \mathrm{B}}{\Gamma \vdash \lambda x{:}\mathrm{A} \,.\, \mathrm{M} : \mathrm{A} \to \mathrm{B}} \ \to+ \qquad \frac{\Gamma \vdash \mathrm{M} : \mathrm{A} \to \mathrm{B} \qquad \Gamma \vdash \mathrm{N} : \mathrm{A}}{\Gamma \vdash \mathrm{M}\,\mathrm{N} : \mathrm{B}} \ \to-$$

Product Types:

$$\frac{\Gamma \vdash \mathrm{M} : \mathrm{A} \qquad \Gamma \vdash \mathrm{N} : \mathrm{B}}{\Gamma \vdash \langle \mathrm{M} \,,\, \mathrm{N} \rangle : \mathrm{A} \times \mathrm{B}} \ \times+ \qquad \frac{\Gamma \vdash \mathrm{M} : \mathrm{A} \times \mathrm{B}}{\Gamma \vdash \pi_0\, \mathrm{M} : \mathrm{A}} \ \times-_0 \qquad \frac{\Gamma \vdash \mathrm{M} : \mathrm{A} \times \mathrm{B}}{\Gamma \vdash \pi_1\, \mathrm{M} : \mathrm{B}} \ \times-_1$$

Unit Type:

$$\frac{}{\Gamma \vdash \star : 1} \ 1+$$

Typing Derivations

A **typing derivation** is a *tree* built from the typing rules where all leaves are rules without premisses:

$$\dfrac{\dfrac{\dfrac{}{x : \mathrm{A} \to \mathrm{A}, y : \mathrm{A} \vdash x : \mathrm{A} \to \mathrm{A}}\ \mathrm{var} \qquad \dfrac{\dfrac{}{x : \mathrm{A} \to \mathrm{A}, y : \mathrm{A} \vdash x : \mathrm{A} \to \mathrm{A}}\ \mathrm{var} \quad \dfrac{}{x : \mathrm{A} \to \mathrm{A}, y : \mathrm{A} \vdash y : \mathrm{A}}\ \mathrm{var}}{x : \mathrm{A} \to \mathrm{A}, y : \mathrm{A} \vdash x\, y : \mathrm{A}}\ {\to}-}{\dfrac{x : \mathrm{A} \to \mathrm{A}, y : \mathrm{A} \vdash x\,(x\, y) : \mathrm{A}}{x : \mathrm{A} \to \mathrm{A} \vdash \lambda y : \mathrm{A} \,.\, x\,(x\, y) : \mathrm{A} \to \mathrm{A}}\ {\to}+}\ {\to}-$$

$$\dfrac{x : \mathrm{A} \to \mathrm{A} \vdash \lambda y : \mathrm{A} \,.\, x\,(x\, y) : \mathrm{A} \to \mathrm{A}}{\vdash \lambda x : \mathrm{A} \to \mathrm{A} \,.\, \lambda y : \mathrm{A} \,.\, x\,(x\, y) : (\mathrm{A} \to \mathrm{A}) \to \mathrm{A} \to \mathrm{A}}\ {\to}+$$

The typing rules have the property that there is a unique rule for each term-forming operation in the conclusion so constitute a **type-checking algorithm** to determine whether a term has a specified type.

Type Inhabitation

The **type inhabitation** problem asks whether a type contains any closed terms.

For example, we can find a term of type $A \times B \to B \times A$:

$$\dfrac{\dfrac{\dfrac{}{x : A \times B \vdash x : A \times B}\,\text{var}}{x : A \times B \vdash \pi_1\, x : B}\,\times\!-_1 \qquad \dfrac{\dfrac{}{x : A \times B \vdash x : A \times B}\,\text{var}}{x : A \times B \vdash \pi_0\, x : A}\,\times\!-_0}{\dfrac{x : A \times B \vdash \langle \pi_1\, x\, ,\, \pi_0\, x \rangle : B \times A}{\vdash \lambda x : A \times B\, .\, \langle \pi_1\, x\, ,\, \pi_0\, x \rangle : A \times B \to B \times A}\,\to\!+}\,\times\!+$$

It's worth thinking about *how* we could find such a term.

In contrast, it's not possible to find a term of type $1 \to A$.

Propositions as Types

Theorem

If we interpret *base types* as *atomic propositions*
and the *type formers* as *logical connectives* as follows:

- 1 as *truth* $\top$,
- $\times$ as *conjunction* $\wedge$,
- $\rightarrow$ as *implication* $\supset$,

then a type is *inhabited* exactly when the corresponding logical proposition is a **tautology**.

So:

- The proposition $A \wedge B \supset B \wedge A$ is a tautology,
- The proposition $\top \supset A$ is not a tautology: it is false whenever A is false.

Terms as Proofs

Theorem

Moreover, we can interpret each *term* of a given *type* as a *proof* of the corresponding *proposition*.

This determines a **constructive logic** known as **intuitionistic logic**.

In intuitionistic logic, proofs are themselves computational objects.

For example, the λ-calculus term $\lambda x : \mathrm{A} \times \mathrm{B} \,.\, \langle \pi_1\, x \,,\, \pi_0\, x \rangle : \mathrm{A} \times \mathrm{B} \rightarrow \mathrm{B} \times \mathrm{A}$

corresponds to an intuitionistic proof of the proposition $\mathrm{A} \wedge \mathrm{B} \supset \mathrm{B} \wedge \mathrm{A}$,

which is an *algorithm* for turning a proof of $\mathrm{A} \wedge \mathrm{B}$ into a proof of $\mathrm{B} \wedge \mathrm{A}$.

Statics and Dynamics

Last time we looked at the **statics** of STLC:

- what are the types?
- what are the terms?
- what type, if any, does a given term have?

Previously, we studied the **dynamics** of pure λ-calculus:

- which terms can a given term β-reduce to?
- does a term normalize?
- if so, what is its normal form?
- do two terms have the same normal form?

Now we turn to the dynamics of STLC.

Typed β-Reduction

Now we have more kinds of β-reducible expressions (redexes).

For *function types* we have redexes like in pure λ-calculus:

$$\overbrace{(\underbrace{\lambda x{:}A\,.\,M}_{\text{abstraction}})\,N}^{\text{application}} \quad \to_\beta \quad \underbrace{M[N/x]}_{\text{substitution}}$$

For *product types* we have redexes for projecting from a pair:

$$\overbrace{\pi_0\,\underbrace{\langle M\,,\,N\rangle}_{\text{pairing}}}^{\text{projection}} \quad \to_\beta \quad M \qquad \text{and} \qquad \overbrace{\pi_1\,\underbrace{\langle M\,,\,N\rangle}_{\text{pairing}}}^{\text{projection}} \quad \to_\beta \quad N$$

These follow a pattern: if we *introduce* a term of function/product type and then *eliminate* it, the result is something simpler.

There is no β-rule for terms of unit type. (Why not?)

As in pure λ-calculus, we can apply β-reductions within subterms.

Subject Reduction

The relation β is **type preserving** in the following sense:

Theorem

For any term well-typed term $\Gamma \vdash M : A$,
if $M \to_{\beta} N$ then $\Gamma \vdash N : A$

This means that if a program computes a result,
then the result will have the same type as the program that computed it.

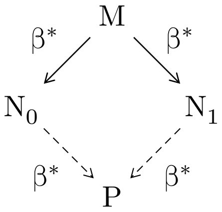
Confluence

Adding types does not interfere with the confluence of β-reduction.

Theorem

The relation β^* is **confluent** in the sense that for any well-typed term $\Gamma \vdash M : A$,

$$\forall N_0, N_1 \,.\, \text{if } M \to_{\beta^*} N_0 \text{ and } M \to_{\beta^*} N_1 \text{ then } \exists P \,.\, N_0 \to_{\beta^*} P \text{ and } N_1 \to_{\beta^*} P.$$



Desiderata of STLC

Recall that the reason for imposing types on λ-calculus was to restrict the admissible terms to those that are *semantically meaningful*.

The β-rules for product and function types ensure that they behave like mathematical products and functions.

We also want to exclude terms that don't have normal forms, like $\Omega := (\lambda x \,.\, x\, x)\,(\lambda x \,.\, x\, x)$.

In this regard STLC is as good as it could possibly be.

Normalization

STLC is **normalizing**:

Theorem (normalization)

For every well-typed term $\Gamma \vdash M : A$
there is a normal term $\Gamma \vdash N : A$ such that $M \downarrow_\beta N$.

Moreover, *every* reduction strategy succeeds in normalizing *any* term:

Theorem (termination)

Every sequence of β-reductions in STLC is finite.

Type Safety

Together, *confluence*, *subject reduction* and *termination* guarantee that any interpreter for STLC will evaluate any well-typed program to a unique value without the possibility of crashing or hanging.

Beyond STLC

There are typed λ-calculi richer than STLC that also have these type safety properties.

Under the Curry-Howard isomorphism we can add type formers corresponding to full first-order logic:

disjunction $\vee$,

falsity $\perp$,

universal quantification $\forall$,

existential quantification $\exists$.

There are several *programming languages* and mathematical *proof assistants* that are designed in this way.