Simply Typed λ-Calculus

Curry-Howard Correspondence

MATH230

Te Kura Pāngarau
Te Whare Wānanga o Waitaha

Outline

Motivation

Type theory was originally formulated by Bertrand Russell and Gottlob Frege as a reaction to set theoretic paradoxes see the Stanford Encyclopaedia article on Type Theory for more on the historical development of the field.

Alonso Church introduced it to the λ-calculus as a way to avoid unending meaningless computation, to ensure that computations don't get stuck.

Consider the following program in the λ-calculus:

$$\lambda f.\, \lambda g.\, \text{COND (EQUAL? } f\ g\text{) (SUCC } f\text{) (SUCC } g\text{)}$$

Typed Languages

In order to get around this Church and others realised the λ-calculus should be augmented with type structure.

In typed λ-calculi each λ-term must be given an explicit type. This requires deciding at the outset the following:

- Base types e.g. Int, Bool, [Int] etc.
- Type constructors i.e. how to build new types from old.

These are language design choices: different type theories (programming languages) will have different base types and allow different type constructors.

In this course we will primarily be interested in type constructors and writing programs, rather than specifying the nature of any particular type. We will follow Type Theory and Functional Programming, Thompson.

Type Declarations

Given a set of base types, $\mathcal{B} = \{A, B, C, \dots\}$

$$\text{A B C} \dots : \text{Type}$$

We must associate any variable $a, b, c, \dots$ in a program to a specific type. We use capital English letters for type-variables and lower-case English letters for term-variables.

We write type declarations as follows:

$x : A$ the term x is of type A

Function Type: Formation

The λ-calculus is closed under abstraction and application, so we need a way to assign types to these λ-terms.

We have been interpreting λ-terms as functions, so we close our set of types under the following operation: if A, B are types, then $(A \to B)$ is also a type, called a function type.

$$\frac{A \ : \ \mathsf{Type} \qquad\qquad B \ : \ \mathsf{Type}}{A \to B \ : \ \mathsf{Type}}$$

Examples

Simply Typed Programs

As in the untyped λ-calculus, programs are written by constructing λ-terms, and composing different λ-terms together.

Now we have to be careful to only create and combine terms according to the typing rules for variables, abstraction, and application.

We will now develop a calculus for programming in the simply typed λ-calculus so that we don't have any type errors.

Type Contexts

We typically write programs in the context of some type declarations. We denote the set of type declarations Σ throughout the notes.

$$\Sigma = \{x : A,\ f : A \to B,\ g : (A \to B) \to C\}$$

Σ might be thought of has global variables, or modules/libraries, that we can use in our programs.

We are interested in a number of questions in relation to the construction of terms, inhabiting particular types, in particular contexts.

Typing Derivation: Variables

If $x : A$ is a type declaration in a type context Σ, then we can always call that term in a program.

$$\Sigma, x : A \vdash x : A$$

$$\frac{}{x : A}\ \text{Var}$$

Function Type: Destructor

If we can write a function type $f : A \to B$ in a type context Σ and we have a term of type $x : A$ in the same context Σ, then we can obtain the term $f\ x : B$ from the same context Σ.

If $\Sigma,\ f : A \to B,\ x : A\ \vdash\ f\ x : B$ by function application.

$$\frac{f : A \to B \qquad\qquad x : A}{(f\ x) : B}\ \text{App}$$

Function Type: Constructor

If we can derive a term $e : B$, which may contain the free variable $a : A$, from the context Σ, then we may *abstract over the* $a : A$, to get the term $(\lambda x : A.\ e) : A \to B$ in the context $\Sigma \backslash \{a : A\}$ *without the declaration* $a : A$.

Note: the term x may appear free in the the body e of the abstraction.

If $\Sigma, x : A \vdash e : B$, then $\Sigma \vdash (\lambda x : A.\ e) : A \to B$

$$\dfrac{\begin{array}{c}\Sigma, \cancel{x\ :\ A} \\ \mathcal{D} \\ e\ :\ B\end{array}}{\lambda x : A.\ e\ \ :\ \ A \to B}\ \lambda 1$$

Well-Typed Terms

We say a λ-term t is well-typed, of type T, in the context Σ if there exists a derivation of t following the typing derivations above which shows $t : T$. We denote this using the notation:

$$\Sigma \ \vdash \ t \ : \ T$$

Example

$$f : A \to B,\ a : A \ \vdash \ (f\ a) \ : \ B$$

Inhabited Types

We a say type T is inhabited relative to a context Σ if there exists a λ-term $t : T$ that can be derived from Σ according to the typing derivations above.

$$\Sigma \;\vdash\; T$$

This is the same idea as the previous slide, now with the emphasis on the type rather than the particular term.

Example

$$f : A \to B,\; a : A \;\vdash\; B$$

I Combinator

Show that the following type is inhabited:

$$\vdash \; P \to P$$

Guiding Questions

We may be tasked with any one of the following questions. If we are given a program, can we show it is well-typed? What context is required to do so? One might instead have a type and want to show that there is a program of that type.

$\Sigma \vdash ? : T$	Type inhabited relative to a context
$\vdash ? : T$	Type inhabited
$\Sigma \vdash t : ?$	Term well-typed relative to a context
$\vdash t : ?$	Term well-typed
$? \vdash t : T$	Find a context
$? \vdash ? : ?$	Find a context with a term of some type

K Combinator

Show that the following type is inhabited:

$$\vdash \; P \to (Q \to P)$$

S Combinator

Prove that the **S** combinator can be well-typed:

$$\mathbf{S} := \lambda x.\ \lambda y.\ \lambda z.\ x\ z\ (y\ z)$$

S Combinator

Prove that the **S** combinator can be well-typed:

$$\mathbf{S} := \lambda x.\ \lambda y.\ \lambda z.\ x\ z\ (y\ z)$$

Example

Show that the following type is inhabited:

$$\vdash\ (A \to B) \to (B \to C) \to (A \to C)$$

Explicit Lambda Types

These λ-terms with their type annotations can be become unwieldy quickly. We can make them shorter by dropping explicit mention of the type of the variable in an abstraction. Since this information is in the type signature of the term, we do not lose anything by doing this.

$$\lambda x : (A \to B \to C).\ \lambda y : (A \to B).\ \lambda z : A.\ x\ z\ (y\ z)\ \ : (A \to B \to C) \to (A \to B) \to A \to C$$

Motivation

William Howard published *The Formulae-as-Types Notion of Construction* in 1980 with the ultimate goal

... to develop a notion of construction suitable for the interpretation of intuitionistic mathematics.

By intuitionistic mathematics, he meant the BHK. His goal was to realise the BHK in a formal system. In so doing he also clarified ideas presented earlier by Haskell Curry (1958) and W. W. Tait (1965).

Brouwer-Heyting-Kolmogorov Interpretation

Recall that intuitionistic proofs of the propositional connectives must be of the following form:

$P \to Q$	to prove an implication we must provide an algorithm for turning a proof of P into a proof of Q
$P \vee Q$	to prove a disjunction we must provide either a proof of P or a proof of Q.
$P \wedge Q$	to prove a conjunction we must provide both a proof of P and a proof of Q.
$\neg P$	to prove a negation we must provide an algorithm that turns a proof of P into a proof of $\perp$

This presentation is taken from the Standford Encyclopedia of Philosophy article by Bridges, Palmgren, and Ishihara.

BHK: Implication

To prove an implication we must provide an **algorithm** for turning a proof of P into a proof of Q.

William Howard (following Curry 1958) knew the simply typed λ-calculus could provide a concrete meaning to the word *algorithm* used in the BHK.

He observed that if the types $P, Q, P \to Q$ of the simply typed λ-calculus were thought of as propositions, then the derivation of a term $f : P \to Q$ is equivalent to an intuitionistic implicational proof of the proposition $P \to Q$.

Moreover, if we interpret a term $x : P$ to represent a proof of P as a proposition, then $f : P \to Q$ really is a function that takes a proof of P and, by beta reduction, will return a proof Q.

Example

$$\vdash \; P \to P$$

$$\dfrac{\overline{P}^{\,1}}{P \to P}\;\to,1 \qquad\qquad \dfrac{\overline{p \;:\; P}^{\,1}}{(\lambda\, x : P.\, x) \;:\; P \to P}\;\lambda,1$$

The **I** combinator is a witness to the fact that $P \to P$ is a tautology.

Example

$$P \ \vdash \ Q \to P$$

$$\dfrac{\dfrac{}{P}}{Q \to P} \to \qquad\qquad \dfrac{\dfrac{}{p \ : \ P}}{(\lambda\ x : Q.\ p) \ : \ Q \to P} \ \lambda$$

We call the term $\lambda x : Q.\ p$ the proof-object of the corresponding natural deduction.

Curry-Howard Correspondence

Curry and Howard's observation is summarised in this table.

Natural Deduction	Simply Typed λ-calculus
Proposition P	Type P
P $\rightarrow$ Q : Prop	P $\rightarrow$ Q : Type
$\rightarrow$ Introduction	λ Abstraction
Modus Ponens	Function Application
Proof of P	Term t of type P

We can state the following concrete metatheorem connecting theorems of positive minimal implicational logic and programs in the simply typed λ-calculus.

Theorem

$$\Sigma \vdash_{\mathsf{PIL}} \alpha \quad \leftrightarrow \quad \Sigma \vdash_{\lambda_{\rightarrow}} t : \alpha$$

This first appeared in Curry (1958) and in a form closer to our notation in Howard (published 1980, manuscript circulated 1969).

Example

$$\vdash \; P \to (Q \to P)$$

$$\dfrac{\dfrac{\overline{\cancel{P}}^{\,1}}{Q \to P}\;\to}{P \to (Q \to P)}\;\to, 1$$

$$\dfrac{\dfrac{\overline{\cancel{p \;:\; P}}^{\,1}}{(\lambda y : Q.\ p) \;:\; Q \to P}\;\lambda}{\lambda x : P.\ (\lambda y : Q.\ x) \;:\; P \to (Q \to P)}\;\lambda, 1$$

The K combinator is a proof-term (witness) of this theorem.

Illustration of Correspondence

$$A \to B, \quad B \to C \vdash A \to C$$

Illustration of Correspondence

Reconstruct the proof corresponding to the following term:

$$\lambda f.\, \lambda g.\, g\ (f\ g)\ :\ ((A \to B) \to A) \to ((A \to B) \to B)$$

λ-terms encode proofs. They <u>are</u> proofs.

Proofs = Programs

Under this correspondence of types-as-propositions, the terms that inhabit the types are often referred to as *proof objects*.

We say a λ-term (program) of type P is a proof object witnessing the proof of P as a proposition.

Question: How does this compare to BHK interpretation of proofs of implication?

Motivation

Curry and Howard showed positive minimal implication proofs correspond to programs in the simply typed λ-calculus.

In his paper, Howard (1980) extended the simply typed λ-calculus to include type constructors and destructors corresponding to the other propositional connectives.

This extended type theory is referred to as *simple type theory*, to distinguish it from dependent type theory. We may also refer to it as *Intuitionistic Type Theory.*

Curry-Howard Correspondence

We will give a type theoretic interpretation of each propositional connective with a type former, constructor, and destructor. Along with computation rules for the corresponding redex.

Logic	Type Theory
Proposition P	Type P
Proof	Program
$\to$	Function type
$\wedge$	
$\vee$	
$\perp$	
$\neg$	

We will follow a modern formulation of these ideas from Type Theory and Functional Programming, Simon Thompson.

Product Type: Formation

Given any two types P, Q we may form another type $P \times Q :$ Type.

$$\frac{A \;:\; \text{Type} \qquad\qquad B \;:\; \text{Type}}{A \times B \;:\; \text{Type}}$$

Examples

Product Type: Constructor

Given inhabited types $p : P$ and $q : Q$ we form a term of the product type as follows:

$$\frac{p \ : \ P \qquad\qquad q \ : \ Q}{(p, q) \ : \ P \times Q} \times$$

To construct a term of type $P \times Q$ a program needs to provide a term $p : P$ **and** a term $q : Q$.

Product Type: Destructor

The product type has two destructors that form terms of the component types. These are FST (FirST) and SND (SecoND).

$$\frac{(a, b) \ : \ A \times B}{\text{fst} \ \ (a, b) \ : \ A}\ \text{fst} \qquad\qquad \frac{(a, b) \ : \ A \times B}{\text{snd} \ \ (a, b) \ : \ B}\ \text{snd}$$

With these new terms we have to say how computations work i.e. what are the β reduction rules for fst , snd ?

$$\text{fst } (a, b) =_\beta a$$

$$\text{snd } (a, b) =_\beta b$$

If these terms appear in any program, then they are β-redex that can be β-reduced according to these equations.

Example

Show that the following type is inhabited:

$$\vdash\ (A \times B) \to (B \times A)$$

$$\dfrac{\dfrac{\dfrac{}{x\ :\ A \times B}\,1}{\mathsf{snd}\ x\ :\ B}\,\mathsf{snd} \qquad \dfrac{\dfrac{}{x\ :\ A \times B}\,1}{\mathsf{fst}\ x\ :\ A}\,\mathsf{fst}}{(\mathsf{snd}\ x, \mathsf{fst}\ x)\ :\ B \times A}\,\times}{(\lambda y : A \times B.\ (\mathsf{snd}\ y)\ (\mathsf{fst}\ y))\ :\ (A \times B) \to (B \times A)}\,\lambda, 1$$

Question: If this proof is a program, then what does it do?

Product is Associative

Show that the following type is inhabited:

$$\vdash \ (A \times B) \times C \to A \times (B \times C)$$

Currying

Show that the following type is inhabited:

$$\vdash \ (A \times B \to C) \to (A \to (B \to C))$$

Coproduct Type: Formation

Given any two types P, Q we may form another type $P + Q$: Type

$$\frac{A \ : \ \text{Type} \qquad\qquad B \ : \ \text{Type}}{A + B \ : \ \text{Type}}$$

Examples

Coproduct Type: Constructors

There are two constructors for the coproduct of two types.

$$\frac{p \ : \ P}{\mathsf{inl}\ \ p \ : \ P + Q}\ \mathsf{inl} \qquad\qquad \frac{q \ : \ Q}{\mathsf{inr}\ \ q \ : \ P + Q}\ \mathsf{inr}$$

To construct a term of type $P + Q$ a program has to provide either a term $p : P$ **or** a term $q : Q$ and tag which one it belongs to.

Coproduct Type: Elimination

The coproduct has one destructor which takes into account the two possible forms a term $x : P + Q$ could take.

$$\frac{x \ : \ P + Q \qquad a \ : \ P \to R \qquad b \ : \ Q \to R}{\text{cases } x \ a \ b \ : \ R} \ \text{cases}$$

There are two possibilities which correspond to whether the term x of $P + Q$ is of the form $x = \text{inl } p$ or $x = \text{inr } q$.

$$\text{cases } (\text{inl } p) \ a \ b =_\beta a \ p$$

$$\text{cases } (\text{inr } q) \ a \ b =_\beta b \ q$$

If these terms appear in any program, then they are β-redex that can be β-reduced according to the these equations. When writing a program with input $t : P + Q$ we have to know what type the element t came from: either P or Q. The two constructors "inl" and "inr" act as tags for the program to know how to treat $t : P + Q$.

Example

Show that the following type is inhabited:

$$\vdash \ (A \times B) \to (A + B)$$

Can you give another proof?

Curry-Howard Correspondence

Logic	Type Theory
Proposition P	Type P
Proof	Program
$\rightarrow$	Function type
$\rightarrow I$	λ
MP	app
$\wedge$	$\times$
$\wedge E_l$	fst
$\wedge E_r$	snd
$\vee$	$+$
$\vee I_l$	inl
$\vee I_r$	inr
$\vee E$	cases

See Type Theory and Functional Programming, Simon Thompson and Naive Type Theory, Thorsten Altenkirch for more details.

Terms and Proofs

Inhabited types of so-called *simple type theory* are therefore in one-to-one correspondence with the theorems of positive minimal logic.

$$\vdash_{\mathsf{STLC}} P \qquad \text{if and only if} \qquad \vdash_{\mathsf{ML}} P$$

Moreover, each natural deduction of a theorem P corresponds to a different λ-term. This isomorphism is not just capturing whether there is a natural deduction, or typing derivation, it is transferring specific deductions to the other side. In this sense the correspondence is proof-relevant. The proofs matter and are transported from logic to type theory and back again.

This correspondence is showing us that the way we break down hypotheses and build conclusions in a proof is the same process as breaking down and building data when writing programs!

Example

Show that the following type is uninhabited:

$$\nvdash (A + B) \rightarrow B$$

Curry-Howard Correspondence

There is a precise correspondence between typed programs and natural deductions in the positive part of minimal logic.

Propositions are types. Proofs are programs. Propositional connectives are type formation rules. Rules of inference are term constructors/destructors.

How far do these ideas extend beyond positive minimal logic? Is there a type corresponding to the proposition $\perp$? Or, the negation type? Can we make type theoretic sense of the propositional rules of inference ex falso and reductio ad absurdrum?

Empty Type: Formation

We add to our type theory the type $\bot$ corresponding to the proposition denoted by the same symbol.

$$\bot \ : \ \text{Type}$$

Following the Curry-Howard correspondence:

$$\vdash t : P \qquad \text{if and only if} \qquad \vdash P$$

If the type $\bot$ is generally inhabited, then it would necessarily be a theorem of propositional minimal logic. It is not a theorem of propositional minimal logic. Therefore, the type $\bot$ should be considered uninhabited. Hence we will refer to it at as the empty type.

In other words, the type $\bot$ has **no introduction rule.**

Empty Type: Destructor

Destructors tell us how to write a program out of the given type. What do we need to provide in order to write a program out of the empty type? Nothing! There is no input to deal with, so we simply return what ever we want!

$$\frac{t \ : \ \bot}{\mathrm{abort}_A(t) \ : \ A} \ \bot\mathrm{E}$$

If a program context Σ allows for the derivation of a term t of type $\bot$, then this rule states that the program can construct a term of any type. This term records the derivation, t, of the type $\bot$ and tags it with “abort” to acknowledge this. That’s all it does. There are no computation rules for the constructor abort_T for any type T. We may think of $\mathrm{abort}_A(t)$ as a record that the program has crashed. The type of an error.

This is the type theoretic interpretation of Ex Falso.

Negation Type

Propositions of the form $\neg P$ were defined in terms of $\bot$ as

$$\neg P :\equiv P \to \bot$$

In our simple type theory then, the type $\neg P$ is defined to be a shorthand for the function type $P \to \bot$.

Example

Determine a proof-object that witnesses a proof of the theorem

$$\vdash \ (P \to Q) \to \neg Q \to \neg P$$

Example

Derive a proof object witnessing the proof of the sequent:

$$\neg A \vee B \ \vdash \ A \to B$$

Example

Derive a proof object witnessing the proof of the sequent:

$$\vdash\ \neg(Q \to P) \to (P \to Q)$$

Curry-Howard Correspondence

We have therefore a type theoretic interpretation of intuitionistic propositional logic. Can this be extended to classical logic?

Logic	Type Theory
Proposition P	Type P
Proof	Program
$\rightarrow$	Function type
$\rightarrow I$	λ
MP	app
$\wedge$	$\times$
$\wedge E_l$	fst
$\wedge E_r$	snd
$\vee$	$+$
$\vee I_l$	inl
$\vee I_r$	inr
$\vee E$	cases
Falsum	Empty Type
XF	$\bot$E

Morally: CHI as BHK

The Curry-Howard Isomorphism can be thought of as a formal system for interpreting the BHK-interpretation of the logical connectives.

Recall the BHK interpretation for proofs of disjunction

$P \vee Q$	to prove a disjunction we must provide either a proof of P or a proof of Q.

If we could extend the propositions-as-types idea to include classical logic, then types of the form $P + \neg P$ would be inhabited for each proposition P. Following the classical proof of LEM given earlier in the course, these proof-objects could not give us a proof befitting the BHK i.e. would not tell us whether the term is of the form inl P or inr $\neg P$.

Continuations

In 1990 Timothy Griffin wrote the paper *A Formulae-as-Types Notion of Control*. In this paper he proves that the programming language *Typed Idealized Scheme* allows for programs to be extracted from classical proofs.

He states:

It is shown that classical proofs possess computational content when the notion of computation is extended to include explicit access to the current control context.

More about this in Chapter 6 of Sørensen and Urzyczyn, *Lectures on the Curry-Howard Isomorphism*.

Double Negation Translation

For a different point-of-view on the computational content of classical theorems, recall that we can characterise the barrier between classical and intuitionistic logic as the use of double negation elimination.

Meta-Theorem: $\Sigma \vdash_C P$ if and only if $\Sigma \vdash_I \neg\neg P$

For any classical theorem P, there is a proof-object witnessing an intuitionistic proof of $\neg\neg P$.

In this sense there is computational content, consistent with the Curry-Howard correspondence, in every classical theorem up-to the point one uses double negation elimination or some other equivalent mode of classical reasoning.

Example

Write a proof-object witnessing the intuitionistic theorem:

$$\vdash \neg\neg(P \vee \neg P)$$

One <u>cannot</u> *refute* the law of excluded middle.

Mathematical Proofs?

Can these languages check proofs of mathematical theorems?

We have seen that first-order predicate logic is enough to express some areas of mathematics and prove mathematical theorems. Therefore, an interpretation of first-order logic in the theory of types would provide a language for doing mathematics where (i) theorem statements are types, and (ii) proofs of those theorems are terms of the corresponding type.

$$\forall x \ : \ \mathbb{N}, \ s(x) = x + 1 \ : \ \text{Type}$$

Per Martin-Löf (student of Andrey Kolmogorov, K of the BHK) and others have developed *dependent type theory* which does this. Moreover modern programming languages have implemented these ideas in what are known as *proof assistants*.

Do Mathematicians Do/Know This?

Mathematicians might be forgiven for not wanting to do natural deductions to prove all their theorems; it is cumbersome and the interesting ideas are often hidden in a sea of uninteresting details.

However, the level of certainty this method provides is a feature that (I believe) should be built into the way we do mathematics. Plus, those that have worked in this direction have claimed to find it an enriching (not to mention fun!) process.

We might be at an inflection point for the rate of adoption of these formal methods.

How to Increase Adoption?

Some of the primary difficulties for adoption of proofs-as-programs:

- Type theory is unusual to mathematicians,
- Lots of boring steps,
- Programming language design, and
- UX Design.

Fundamentally then much of the problem is a software engineering problem!

Full adoption will require the collaboration of mathematicians, programming language theorists, and software engineers. Moves are being made in this direction!

Hilbert's Program

We set out to see what came of Hilbert's Program to determine whether mathematics could be put on firm foundations. To show mathematics is:

- Complete,
- Consistent,
- Decidable by an algorithm.

Gödel showed that PA is not completable and that there can be no finitary proof that PA is consistent.

Turing and Church showed that decidability by an algorithm is impossible for theorems of first-order logic.

All of Hilbert's questions were answered in the negative...

Unexpected Landmarks

... But a great number of interesting ideas were developed along the way that have had an effect on mathematics and society.

These theoretical developments have all happened in parallel with the engineering of machines that can carry out computation at a great scale. Many of the mathematical models of computation fed into both the design of machines and the design of the languages we use to talk to the machines!

Although Turing and Church showed that we can't have an algorithm for deciding theoremhood, we do now have machines that can help us write and check the proofs of theorems. Moreover, parts of this process <u>can</u> be automated by these machines.

Further Reading

This lecture was prepared with the aid of the following references. These should be consulted for further detail on the topics.

Stanford Encyclopaedia of Philosophy Articles: Type Theory, Intuitionistic Type Theory, and Constructive Mathematics.

Type Theory and Functional Programming, Simon Thompson

Naive Type Theory, Thorsten Altenkirch

Each of these are freely available on the world wide web.