

Typed λ -Calculi

Curry-Howard Correspondence

MATH230

Te Kura Pāngarau
Te Whare Wānanga o Waitaha

- ① Type Theory
- ② Curry-Howard Correspondence
- ③ Simple Type Theory
- ④ Proof Assistants
- ⑤ Course Summary

Type theory was originally formulated by Bertrand Russell and Gottlob Frege as a reaction to set theoretic paradoxes [sep-type-theory].

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

Consider the following program in the λ -calculus:

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

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.

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 **[thompson]**.

Type Declarations

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

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 $(A \rightarrow B)$ is also a type, called a function 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.

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

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

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.

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

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

Typing Derivation: Application

If we can write a function type $f : A \rightarrow B$ 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 $\Sigma, f : A \rightarrow B, x : A \vdash f x : B$ by function application.

$$\frac{f : A \rightarrow B \quad x : A}{(f x) : B} \text{ App}$$

Typing Derivation: Abstraction

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 $(\lambda x : A. e) : A \rightarrow B$ in the context $\Sigma \setminus \{a : A\}$ *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 $\Sigma \vdash (\lambda x : A. e) : A \rightarrow B$

$$\frac{\begin{array}{c} \cancel{[a : A]^+} \\ \vdots \\ e : B \end{array}}{(\lambda x : A. e) : A \rightarrow B} \lambda, 1$$

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

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

We 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.

$$\Sigma \vdash t : T$$

Example

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

Show that the following type is inhabited:

$$\vdash ? : P \rightarrow P$$

Guiding Questions

We may be tasked with any one of the following questions. If we are given a program, can we show it's 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

Example

Show that the following type is inhabited in the given context:

$$p : P \vdash ? : (Q \rightarrow P)$$

Example

Show that the following type is inhabited in the given context:

$$f : A \rightarrow (B \rightarrow C) \vdash ? : ((A \rightarrow B) \rightarrow (A \rightarrow C))$$

Explicit Lambda Types

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

William Howard [Howard1980] 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.

However, 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:

$P \rightarrow Q$	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 Stanford Encyclopedia of Philosophy article by Bridges, Palmgren, and Ishihara [[sep-mathematics-constructive](#)].

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

Hello Maia this a change to the file.

$$\vdash P \rightarrow P$$

$$\frac{\overline{P} \ 1}{P \rightarrow P} \rightarrow, 1$$

$$\frac{\overline{p : P} \ 1}{(\lambda x : P. x) : P \rightarrow P} \lambda, 1$$

$$P \vdash Q \rightarrow P$$

$$\frac{\overline{P}}{Q \rightarrow P} \rightarrow$$

$$\frac{\overline{p : P}}{(\lambda x : Q. p) : Q \rightarrow P} \lambda$$

Curry-Howard Correspondence

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

Natural Deduction	Simply Typed λ -calculus
Proposition P	Type P
\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 implicative logic and programs in the simply typed λ -calculus.

Theorem

$$\Sigma \vdash_{IPL} \alpha \quad \leftrightarrow \quad \Sigma \vdash_{\lambda \rightarrow} t : \alpha$$

This first appeared in Curry (1958) [**curryCombinatoryLogic**] and in a form closer to our notation in Howard (1980).

$$\vdash P \rightarrow (Q \rightarrow P)$$

$$\frac{\frac{\overline{\cancel{P}}^1}{Q \rightarrow P} \rightarrow}{P \rightarrow (Q \rightarrow P)} \rightarrow, 1$$

$$\frac{\frac{\overline{\cancel{p : P}}^1}{(\lambda y : Q. p) : Q \rightarrow P} \lambda}{\lambda x : P. (\lambda y : Q. x) : P \rightarrow (Q \rightarrow P)} \lambda, 1$$

Example

$$A \rightarrow B, B \rightarrow C \vdash A \rightarrow C$$

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?

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

In his paper, Howard extended the simply typed λ -calculus to include type constructors and destructors corresponding to the other propositional connectives [**Howard1980**].

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

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

We will follow a modern formulation of these ideas from Simon Thompson's *Type Theory and Functional Programming* [thompson].

Product Type: Formation

Given any two types P, Q we may form another type

$$P \times Q$$

We close our set of types under this operation.

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

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

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: Elimination

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

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

Product Type: Computation

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.

Show that the following type is inhabited:

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

$$\frac{\frac{\overline{x : A \times B} \quad 1}{\text{snd } x : B} \text{ snd} \quad \frac{\overline{x : A \times B} \quad 1}{\text{fst } x : A} \text{ fst}}{(\text{snd } x, \text{fst } x) : B \times A} \times \quad \frac{}{(\lambda y : A \times B. (\text{snd } y) (\text{fst } y)) : (A \times B) \rightarrow (B \times A)} \lambda, 1$$

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

Show that the following type is inhabited:

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

Coproduct Type: Formation

Given any two types P, Q we may form another type

$$P + Q$$

We close our set of types under this operation.

Given types P, Q we construct a term of the coproduct $P + Q$ given either a term $p : P$ or a term $q : Q$.

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

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.

$$\frac{x : P + Q \quad a : P \rightarrow R \quad b : Q \rightarrow R}{\text{cases } x \text{ a } b : R} \text{ cases}$$

Coproduct Type: Computation

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 $t : P + Q$.

Show that the following type is inhabited:

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

Can you give another proof?

Curry-Howard Correspondence

Logic	Type Theory
Proposition P	Type P
Proof	Program
\rightarrow	Function type
\wedge	\times
$\wedge E_l$	FST
$\wedge E_r$	SND
\vee	$+$
$\vee I_l$	inl
$\vee I_r$	inr
$\vee E$	cases

See Simon Thompson *Type Theory and Functional Programming* [**thompson**] and Thorsten Altenkirch *Naive Type Theory* [**naiveTT**] for more details.

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

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

Show that the following type is uninhabited:

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

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 absurdum?

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

Following the Curry-Howard correspondence:

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

if the type \perp 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 \perp should be considered generally uninhabited.

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

Negation Type

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

$$\neg P :\equiv P \rightarrow \perp$$

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

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

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

Although there is no introduction rule for \perp , we know that contradictory contexts may still derive terms of type \perp . Just as contradictory hypotheses can derive the False proposition \perp .

In some programs/proofs the \perp type/proposition gets tucked away in a negation using λ /implication introduction. As in the previous example.

Although in other proofs there are rules of inference relating to the elimination of \perp by other means. Namely, *ex falso* and *reductio ad absurdum*. What can we make of these logical rules on the side of type theory?

The \perp elimination rule follows from the *ex falso* rule of inference.

$$\frac{t : \perp}{\text{abort}_A(t) : A} \perp\text{E}$$

If a program context allows for the derivation of a term t of type \perp , then this rule states that the program can construct a term of any type.

This term records the derivation, t , of the type \perp 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 $\text{abort}_A(t)$ as a record that the program has crashed. The type of an error.

Derive a proof object witnessing the proof of the sequent:

$$\neg A \vee B \vdash A \rightarrow B$$

Derive a proof object witnessing the proof of the sequent:

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

Curry-Howard for Classical Logic

The following slides are a discussion about extending these ideas from the intuitionistic setting they were founded in to classical propositional logic.

Extending in this way has consequences for the interpretation of the proof-objects obtained.

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 $P \vee \neg P$ 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 which of P or $\neg P$ is true.

In 1990 Timothy Griffin wrote the paper *A Formulae-as-Types Notion of Control* [griffin]. 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 [LoCHI].

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. The following meta-theorem states exactly that:

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.

Write a proof-object witnessing the intuitionistic theorem:

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

One cannot *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.

$$\forall x \ s(x) = x + 1$$

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.

Do Mathematicians Do/Know This?

Mathematicians can be forgiven for not wanting to do natural deductions to prove all their theorems; it's 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.

Metamathematical Landmarks

Along the way first-order logic and two ideas about proof and consequence were developed. Gödel's completeness theorem showed them to be equivalent.

Gödel then 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 developments have all happened alongside 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, large parts of this process can now 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.