



1 Warmup Questions

1. In the calculus of truth and falsehood, define Γ , δ , proofs, refutations, contradictions and continuations.

Definition 1 (Γ). Γ is the truth context – it is a context of the form $\alpha, \beta, \gamma, \dots$ which contains only things we assume to be true.

Definition 2 (Δ). Δ is the false context – it is a context of the form $\alpha, \beta, \gamma, \dots$ which contains only things we assume to be false.

Definition 3 (Proof). A proof is a derivation which concludes something of the form $\Gamma, \Delta \vdash A$ true. This means “if the assumptions Γ are true, and the assumptions Δ are false, then we have that A is true”.

Definition 4 (Refutation). A proof is a derivation which concludes something of the form $\Gamma, \Delta \vdash A$ false. This means “if the assumptions Γ are true, and the assumptions Δ are false, then we have that A is false”.

Definition 5 (Contradiction). A contradiction occurs when we can prove that something is both true and false. They are of the form $\Gamma, \Delta \vdash A$ **contr** and we can use these to conclude that an arbitrary assumption in Γ is false; or symmetrically that an arbitrary assumption in Δ is true.

Definition 6 (Continuation). A continuation is a term k which we have proved has the type A false for some type A .

2. What makes dependent types so powerful? Give an example of their usefulness in everyday programming.

Dependent types are so powerful because arbitrary programs are now types. This means that the type system is far more expressive. The type system can now be used for proofs. This is useful when writing safety critical code. Additionally, the type system is now able to detect terms which have incompatible values at runtime.

Consider zipping two generators of different lengths together *i.e.* `zip(range(10), range(20))`. In a language without dependent types, this will work and the program will error at runtime after the first generator runs out of items. However, with dependent types, this is now a type error (since the length of the generators is different and `zip` would require them to be the same).

3. What are the downsides of dependent types?
 - More complicated rules – much harder to understand
 - Longer compilation time
 - Hard to prove properties of the language
4. Write down and carefully explain (in English) the subst rule in the dependent types part of the course.

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma, x : A \vdash B \text{ type} \quad \Gamma \vdash p : (e_1 = e_2 : A) \quad \Gamma \vdash e : [e_1/x]B}{\Gamma \vdash \text{subst}[x : A. B](p, e) : [e_2/x]B}$$

This rule allows us to replace a term, e_1 occurring in the type $[e_1/x]B$ of a term e with another term e_2 if we have proved they are equal. The premise ensure that e_1 and e_2 are typed to a derivable term, that we do indeed have a proof that $(e_1 = e_2 : A)$ is a valid type.





2 Regular Questions

1. Complete Exercises 1 and 2 from Lecture 9:

(a) Show that $\neg A \vee B, A; \cdot \vdash B$ true is derivable.

$$\begin{array}{c}
 \frac{}{\neg A \vee B, A; B \vdash \neg A \vee B \text{ true}} \quad \frac{}{\neg A \vee B, A; B \vdash B \text{ false}} \quad \frac{}{\neg A \vee B; \neg A, B \vdash \neg A \text{ false}} \\
 \frac{}{\neg A \vee B, A; B \vdash \neg A \text{ false}} \\
 \frac{}{\neg A \vee B, A; B \vdash \neg A \vee B \text{ false}} \\
 \frac{}{\neg A \vee B, A; B \vdash \text{contr}} \\
 \frac{}{\neg A \vee B, A; \cdot \vdash B \text{ true}}
 \end{array}$$

Figure 1: Proof that $\neg A \vee B, A; \cdot \vdash B$ true is derivable

(b) Show that $\neg(\neg A \wedge \neg B); \cdot \vdash A \vee B$ true is derivable

$$\begin{array}{c}
 \frac{}{\neg(\neg A \wedge \neg B); \cdot \vdash \neg(\neg A \wedge \neg B) \text{ true}} \\
 \frac{}{\neg(\neg A \wedge \neg B); \cdot \vdash \neg A \wedge \neg B \text{ false}} \\
 \frac{}{\neg(\neg A \wedge \neg B); \cdot \vdash \neg A \text{ false}} \\
 \frac{}{\neg(\neg A \wedge \neg B); \cdot \vdash A \text{ true}} \\
 \frac{}{\neg(\neg A \wedge \neg B); \cdot \vdash A \vee B \text{ true}}
 \end{array}$$

Figure 2: Proof that $\neg(\neg A \wedge \neg B); \cdot \vdash A \vee B$ true is derivable

2. Give the proof (and refutation) terms corresponding to the derivations in the previous question.

$$x : \neg A \vee B, y : A; \cdot \vdash \mu u : B. \langle x \mid_B [\text{not}(y), \text{not}(u)] \rangle : B \text{ true}$$

$$x : \neg(\neg A \wedge \neg B); \cdot \vdash \text{not}(\text{fst}(\text{not}(x))) : A \vee B \text{ true}$$

3. Let $\text{upc}(p) \triangleq \mu u : A. \langle p \mid_A u \rangle$ be a proof (and refutation) term from the calculus presented in Lectures 9 and 10.

(a) Show that $p : A; \cdot \vdash \text{upc}(p) : A$ true.

$$\begin{array}{c}
 \frac{}{p : A, u : A \vdash p : A \text{ true}} \quad \frac{}{p : A, u : A \vdash u : A \text{ false}} \\
 \frac{}{p : A, u : A \vdash \langle p \mid_A u \rangle : \text{contr}} \\
 \frac{}{p : A; \cdot \vdash \mu u : A. \langle p \mid_A u \rangle : A \text{ true}} \\
 \frac{}{p : A; \cdot \vdash \text{upc}(p) : A \text{ true}}
 \end{array}$$

Figure 3: Derivation tree for $\text{upc}(p)$

(b) Show that for all $k : \neg A$, we have $\langle \text{upc}(p) \mid_A k \rangle \mapsto \langle p \mid_A k \rangle$.

We have that $\text{upc}(p) \triangleq \mu u : A. \langle p \mid_A u \rangle$, we have that *forall* $k : \neg A. \langle \text{upc}(p) \mid_A k \rangle = \langle \mu u. c \mid_A k \rangle$. This is the premise of one of the reduction rules. Thus, we can reduce this into the form $[k/u] \langle p \mid_A k \rangle = \langle p \mid_A k \rangle$. So, we have that $\forall k : \neg A. \langle \text{upc}(p) \mid_A k \rangle \mapsto \langle p \mid_A k \rangle$.



- (c) Terms $p : A$ true correspond to proofs of A . Describe, in English, the proof that corresponds to $\text{upc}(p)$ with respect to the proof corresponding to p .

$\text{upc}(p)$ corresponds to a proof by contradiction for A which first proves A , then a space where a refutation for A would be; followed by a contradiction since A cannot both hold and be refuted – thus we can conclude that A must hold!

4. Complete Exercises 1 and 2 from Lecture 10:

- (a) Give the embedding (*i.e.* the e° and k° translations) of classical into intuitionistic logic for the Gödel translation. You just need to give the embeddings for sums, since this is the only case different from the lectures.

$$\begin{aligned}(L e_1)^\circ &= \lambda k : (\sim X^\circ \times \sim Y^\circ). (\text{fst } k) e_1 \\ (R e_2)^\circ &= \lambda k : (\sim X^\circ \times \sim Y^\circ). (\text{snd } k) e_2\end{aligned}$$

- (b) Using the intuitionistic ($\lambda-$) calculus extended with continuations, give a typed term proving Peirce's law:

$$((X \rightarrow Y) \rightarrow X) \rightarrow X$$

The term below has the desired type:

$$\lambda f : (X \rightarrow Y) \rightarrow X. \text{letcont } k : \sim X. f(k)$$

I prove this in [Figure 4](#).

5. 3 2021 Paper 8 Question 15

- (a) In the calculus of proofs and refutations, suppose that $\Gamma; \Delta \vdash A$ true and $\Gamma, A; \Delta \vdash C$ true. Show that $\Gamma; \Delta \vdash C$ true is derivable. [*Hint*: Recall that weakening is admissible in this calculus]

I prove this in [Figure 5](#).

- (c) Consider the following piece of Agda code, where `Nat` is the type of natural numbers:

```
X : (P : Nat -> Set) ->
  P 0 ->
  ((n : Nat) -> P n -> P (1 + n)) ->
  (k : Nat) -> P k
X P base step zero = base
X P base step (suc n) = step n (X P base step n)
```

- (i) Explain what the `X` function means in logical terms.

`X` takes two assumptions $P\ 0$ and $P\ n \implies P\ (1 + n)$ and applies modulus ponens k times to provide a prove by induction that $P\ k$ must be true.

- (ii) Explain what the `X` function does in terms of functional programming.

In terms of functional programming, `X` is an abstract interface for the natural numbers: it takes a function from natural numbers to types, a zero type and a successor function and a natural number k and returns the value representing the natural number k .



<https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2021p8q15.pdf>



<https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2019p8q13.pdf>

6. 4 2019 Paper 8 Question 13

Recall the three judgements for classical propositional logic:

- (a) $\Gamma; \Delta \vdash e : A \text{ true}$ – e is a proof of type A .
- (b) $\Gamma; \Delta \vdash k : A \text{ false}$ – k is a refutation of type A .
- (c) $\Gamma; \Delta \vdash \langle e \mid_A k \rangle \text{ contr}$ – $\langle e \mid_A k \rangle$ is a contradiction at type A .

Here, Γ contains all the true assumptions, and Δ are the false assumptions. In this question, we will extend classical propositional logic with support for the implication or function type operator $A \rightarrow B$.

- (a) Give a proof term and inference rule for a proof of type $A \rightarrow B$.

The proof terms I introduce to represent this are $e \rightarrow$ and $\rightarrow k$. The function type $A \rightarrow B$ is represented by the type $\neg A \vee B$ and therefore it should be “syntactic sugar” for the disjunction of those two types.

$$\frac{\overline{\Gamma; \Delta \vdash k : A \text{ false}}}{\Gamma; \Delta \vdash k \rightarrow : A \rightarrow B \text{ true}}$$

Figure 6: A Typing Rule for a proof of $A \rightarrow B$ true

- (b) Give a proof term and inference rule for a refutation of type $A \rightarrow B$.

The proof term I introduce to represent a refutation of type $A \rightarrow B$ is $e \text{ not } \rightarrow k$.

$$\frac{\overline{\Gamma; \Delta \vdash e : A \text{ true}} \quad \overline{\Gamma; \Delta \vdash k : A \text{ false}}}{\Gamma; \Delta \vdash e \text{ not } \rightarrow k : A \rightarrow B \text{ false}}$$

Figure 7: A Typing Rule for a refutation of $A \rightarrow B$ false

- (c) Give a reduction rule for contradiction configurations of the form $\langle e \mid_{A \rightarrow B} k \rangle$.

$$\langle k_1 \rightarrow \mid_{A \rightarrow B} e \rightarrow k_2 \rangle \mapsto \langle e \mid_A k_1 \rangle$$

- (d) (i) State the preservation theorem for classical logic.

$$\Gamma; \Delta \vdash c \text{ contr} \quad \wedge \quad c \mapsto c' \quad \implies \quad \Gamma; \Delta \vdash c' \text{ contr}$$

- (ii) Give the proof of preservation for the case of the new rule defined above. You may assume that weakening, exchange and substitution all hold.

Case $\langle k_1 \rightarrow \mid_{A \rightarrow B} e \rightarrow k_2 \rangle \mapsto \langle k_1 \mid_A e \rangle$

$\langle k_1 \rightarrow \mid_{A \rightarrow B} e \rightarrow k_2 \rangle \mapsto \langle k_1 \mid_A e \rangle$	Assumption	(1)
$\Gamma; \Delta \vdash \langle k_1 \rightarrow \mid_{A \rightarrow B} e \rightarrow k_2 \rangle \text{ contr}$	Assumption	(2)
$\Gamma; \Delta \vdash k_1 : A \text{ false}$	Inversion	(3)
$\Gamma; \Delta \vdash k_2 : B \text{ false}$	Inversion	(4)
$\Gamma; \Delta \vdash e : A \text{ true}$	Inversion	(5)
$\Gamma; \Delta \vdash \langle e \mid_A k_2 \rangle \text{ contr}$	Typing Rule on (3) and (5)	(6)



7. Using the `amb` primitive from Lecture 11 implement a function:

`eq-at : α list -> α list -> int * int`

such that `eq-at xs ys` returns `(i, j)` if `nth(xs, i) = nth(ys, j)` and fails otherwise. You may assume the existence of any helper functions without definition.

```
let eq-at xs ys =
  let i = amb(list(range(len(xs))) in
  let j = amb(list(range(len(ys))) in
  assert nth (xs, i) == nth (ys, j)
  (i, j)
```

8. Using the dependent type theory introduced in Lecture 12 show that if $\Gamma \vdash A$ type then the following typing judgement holds:

$$\Gamma \vdash \text{sym}_A : \Pi x : A. \Pi y : A. ((x = y : A) \rightarrow y = x : A)$$

where

$$\text{sym}_A \triangleq \lambda x : A. \lambda y : A. \lambda p : (x = y : A). \text{subst}[z : A. (z = x : A)](p, \text{refl } x)$$

and $X \rightarrow Y$ is shorthand for $\Pi x : X. Y$ if x does not appear in Y .

With $\Gamma' = \Gamma, x : A, y : A, p : (x = y : A)$

9. Define terms with the following types:

(a) $\Gamma \vdash \text{trans}_A : \Pi x : A. \Pi y : A. \Pi z : A. ((x = y : A) \rightarrow (y = z : A) \rightarrow (x = z : A))$

$$\text{trans}_A \triangleq \lambda x : A. \lambda y : A. \lambda z : A. \lambda p : (x = y : A). \lambda q : (y = z : A). \text{subst}[y : A. (x = z : A)](q, p)$$

(b) $\Gamma \vdash \text{cong}_{A,B} : \Pi x : A. \Pi y : A. \Pi f : (A \rightarrow B). ((x = y : A) \rightarrow (f x = f y : B))$

$$\text{cong}_{A,B} \triangleq \lambda x : A. \lambda y : A. \lambda f : (A \rightarrow B). \lambda p : (x = y : A). \text{subst}[z : A. (f x = f z)](p, \text{refl } f x)$$

Assuming that $\Gamma \vdash A$ type and $\Gamma \vdash B$ type

5 Extension Questions

- Download and install Agda and try out some of the examples from the lectures:
Haven't but will eventually... This has been on my TODO list for half a year.
- Consider types $\Gamma \vdash A$ type and $\Gamma, x : A \vdash B$ type. if we have terms a_1 and a_2 and a proof that they are equal, $\Gamma \vdash p : (a_1 = a_2 : A)$, then the types $[a_1/x]B$ and $[a_2/x]B$ should also be "equal" in some sense. And so, given $\Gamma \vdash b_1 : [a_1/x]B$ and $\Gamma \vdash b_2 : [a_2/x]B$ we might want to consider the type of equalities between b_1 and b_2 .

- (a) Show that the following rule is not (in general) derivable:

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma, x : A \vdash B \text{ type} \quad \Gamma \vdash p : (a_1 = a_2 : A) \quad \Gamma \vdash b_1 : [a_1/x]B \quad \Gamma \vdash b_2 : [a_2/x]B}{\Gamma \vdash (b_1 = b_2 : [a_1/x]B) \text{ type}}$$



I prove this by counterexample:

$$\begin{aligned}
 A &= 1 \\
 B &= x \\
 \Gamma &= p : (\langle \rangle = \lambda y : 1. \langle \rangle : 1) \\
 a_1 &= \langle \rangle \\
 a_2 &= \lambda y : 1. \langle \rangle \\
 b_1 &= 1 \\
 b_2 &= \lambda y : 1. y \\
 p &= (\langle \rangle = \lambda y : 1. \langle \rangle : 1)
 \end{aligned}$$

$$\begin{array}{ll}
 p : (\langle \rangle = \lambda y : 1. \langle \rangle : 1) \vdash (\langle \rangle = \lambda y : 1. y : 1) \text{ type} & \text{assumption} \\
 p : (\langle \rangle = \lambda y : 1. \langle \rangle : 1) \vdash \langle \rangle : 1 & \text{inversion} \\
 p : (\langle \rangle = \lambda y : 1. \langle \rangle : 1) \vdash \lambda y : 1. y : 1 & \text{inversion}
 \end{array}$$

By inspection, we can see that there is no way to derive $\lambda y : 1. y : 1$ and therefore the proof cannot proceed. So the rule does not hold in general.

(b) Define the type of *heterogeneous equalities* like so:

$$(b_1 \approx b_2 : B \text{ over } p) \triangleq (\text{subst}[x : A. B](p, b_1) = b_2 : [a_2/x]B)$$

Show that the following rule is derivable:

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma, x : A \vdash B \text{ type} \quad \Gamma \vdash p : (a_1 = a_2 : A) \quad \Gamma \vdash b_1 : [a_1/x]B \quad \Gamma \vdash b_2 : [a_2/x]B}{\Gamma \vdash (b_1 \approx b_2 : B \text{ over } p) \text{ type}}$$

(c) Define a term $\text{hrefl } b$ such that the following rule is derivable:

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma, x : A \vdash B \text{ type} \quad \Gamma \vdash a : A \quad \Gamma \vdash b : [a/x]B}{\Gamma \vdash \text{hrefl } b : (b \approx b : B \text{ over } (\text{refl } a))}$$

Notice that we can use judgemental term equality to derive that any term with the type $(b = b : A)$ has the type required. Thus, $\text{hrefl } b$ is an arbitrary term with type $(b = b : A)$. I choose:

$$\text{hrefl } b = \text{refl } b$$

We can then use the rules with conclusions $\Gamma \vdash (e_1 = e_2 : A) \equiv (e'_1 = e'_2 : A') \text{ type}$ and $\Gamma \vdash \text{subst}[x : A. B](e_1, e_2) \equiv \text{subst}[x : A'. B'](e'_1, e'_2 : [e'/x]B)$ to derive that $(b = b : A) \text{ type} \equiv \text{hrefl } b : (b \approx b : B \text{ over } (\text{refl } a))$.

We can then use the judgemental equality rule to derive that $\text{hrefl } b$ also has this type.

3. Didn't find a nice answer.

6 Practice Material

1. Is there a way of constructing a sequence of terms t_1, t_2, \dots in the simply typed λ -calculus with only the base type 1, such that, for each n , the term t_n has size at most $\mathcal{O}(n)$, but requires $\mathcal{O}(2^n)$ steps of evaluation to reach a normal form?



Yes, there is. I propose the set of terms t_i defined by:

$$\begin{aligned} f &= \lambda f y. ((\text{fst } f y)(\text{snd } f y), (\text{fst } f y)(\text{snd } f y)) \\ x_1 &= (\lambda y. y, \langle \rangle) \\ x_{i+1} &= (f, x_i) \\ t_i &= f x_i \end{aligned}$$

We see that $t_{n+1} \rightsquigarrow^+ (t_n, t_n)$. So the evaluation time is $\mathcal{O}(2^n)$. Since $t_n = f \underbrace{(f(f, \dots x_1))}_{n \times}$, the size of t_n is $\mathcal{O}(n)$.

2. Show that if a term e is typeable in the empty context, then e must be *closed*, i.e. have no free variables.

Didn't get around to this.

3. (a) Write a function `isnil` that takes a Church-encoded list and returns a Church boolean.

$$\begin{aligned} \text{isnil} &\triangleq \lambda l. l \text{ bool true } (\lambda x : X. \text{false}) \\ \text{bool} &\triangleq \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \\ \text{true} &\triangleq \Lambda \alpha. \lambda x : \alpha. \lambda y : \alpha. x \\ \text{false} &\triangleq \Lambda \alpha. \lambda x : \alpha. \lambda y : \alpha. y \end{aligned}$$

`isnil` iterates through the list. The list is initially passed `true` – and a function which returns `false`. So if the list is empty then it returns `true` else it returns `false` – as required.

- (b) Write a function `head` : $X \rightarrow \text{CList}_X \rightarrow X$ which returns the first argument if the list is empty and the head of the list otherwise.

$$\text{head} \triangleq \lambda x. \lambda l. l X x (\text{true } X)$$

`head` iterates through the list (starting at the end) and keeps only the most recent element its seen. This means that it will keep the head of the list – or x if the list is empty.

- (c) Write a function `tail` : $X \rightarrow \text{CList}_X \rightarrow \text{CList}_X$ which returns the empty list when applied to an empty list, and the tail of the list otherwise.

$$\begin{aligned} \text{List} &\triangleq \forall \alpha. \alpha \rightarrow (X \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \\ [] &\triangleq \text{true} \\ a :: l &\triangleq \Lambda A. \lambda x. \lambda f. \\ X + Y &\triangleq \forall \alpha. (X \rightarrow \alpha) \rightarrow (Y \rightarrow \alpha) \rightarrow \alpha \\ L e &\triangleq \Lambda \alpha. \lambda f. \lambda g. f e \\ R e &\triangleq \Lambda \alpha. \lambda f. \lambda g. g e \\ X \times Y &\triangleq \forall \alpha. (X \rightarrow Y \rightarrow \alpha) \rightarrow \alpha \\ (e, e') &\triangleq \Lambda \alpha. \lambda f. f e e' \\ \text{fst } p &\triangleq p X (\lambda x. \lambda y. x) \\ \text{snd } p &\triangleq p X (\lambda x. \lambda y. y) \\ \text{tail} &\triangleq \lambda l. (l (\text{List} + (\text{List} \times X)) (L [])) \\ &\quad (\lambda x. \lambda l. l (\text{List} + (\text{List} \times X)) (\lambda y. R ([], y)) (\lambda y. ((\text{snd } y) :: (\text{fst } y)))) \\ &\quad \text{List } (\lambda l. l) (\lambda l. \text{fst } l) \end{aligned}$$



4. Suppose that $\Theta, \beta \vdash F$ type, and `fmap` is a term such that:

$$\Theta; \cdot \vdash \text{fmap} : \forall \beta_1. \forall \beta_2. (\beta_1 \rightarrow \beta_2) \rightarrow ([\beta_1/\beta]F \rightarrow [\beta_2/\beta]F)$$

The type μF is defined to be $\forall \beta. (F \rightarrow \beta) \rightarrow \beta$.

- (a) Show that $\Theta \vdash \mu F$ type and $\Theta \vdash [\mu F/\beta]F$ type.

$$\frac{\frac{\frac{\Theta, \beta \vdash F \text{ type}}{\Theta, \beta \vdash F \rightarrow \beta \text{ type}} \quad \frac{\Theta, \beta \vdash \beta \text{ type}}{\Theta, \beta \vdash \beta \text{ type}}}{\Theta, \beta \vdash (F \rightarrow \beta) \rightarrow \beta \text{ type}}}{\Theta \vdash \forall \beta. (F \rightarrow \beta) \rightarrow \beta \text{ type}}$$

Figure 8: Proof tree to show $\Theta \vdash \mu F$ type

$$\frac{\frac{\Theta \vdash \mu F \text{ type}}{\Theta \vdash \mu F \text{ type}} \quad \frac{\Theta, \beta \vdash F \text{ type}}{\Theta \vdash [\mu F/\beta]F \text{ type}}}{\Theta \vdash [\mu F/\beta]F \text{ type}}$$

Figure 9: Proof tree to show $\Theta \vdash [\mu F/\beta]F$ type

- (b) Define terms `fold` and `intro` such that:

$$\Theta; \cdot \vdash \text{fold} : \forall \beta. (F \rightarrow \beta) \rightarrow \mu F \rightarrow \beta$$

$$\Theta; \cdot \vdash \text{intro} : [\mu F/\beta]F \rightarrow \mu F$$

$$\text{fold} \triangleq \Lambda \beta. \lambda f. \lambda g. g \beta f$$

$$\text{intro} \triangleq \lambda x : [\mu F/\beta]F. \Lambda \beta. \lambda f : F \rightarrow \beta. f(\text{fmap} (\lambda g : \mu F. g \beta f) x)$$

5. 7 2022 Paper 9 Question 13

- (a) Using the simply-typed lambda calculus with the `letcont` primitive, give well-typed terms corresponding to proofs of the following classical tautologies:

- (i) $\text{dne} : \neg \neg A \rightarrow A$

$$\text{dne} \triangleq \lambda x : \neg \neg A. \text{letcont } u : \neg X. \text{throw}(x, u)$$

- (ii) $\text{contr} : (A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$

$$\text{contr} \triangleq \lambda f. \lambda b : \neg B. \lambda a : A. \text{throw}(b, f a)$$

- (iii) $\text{demorgan}_1 : \neg(A \vee B) \rightarrow \neg A \wedge \neg B$

$$\text{demorgan}_1 \triangleq \lambda x : \neg(A \vee B). (\lambda a : A. x(L a), \lambda b : B. x(R b))$$

- (iv) $\text{demorgan}_2 : (\neg A \wedge \neg B) \rightarrow \neg(A \vee B)$

$$\text{demorgan}_2 \triangleq \lambda x : \neg A \wedge \neg B. \lambda y : A \vee B. \text{case}(y, L a \rightarrow (fst x) a, R b \rightarrow (snd x) b)$$

- (b) (i) Briefly explain what the following Agda type says.

$$\forall \{A : \text{Set}\} \{n : \text{Nat}\} \rightarrow \text{Vec } A \ n \rightarrow (i : \text{Nat}) \rightarrow (i < n) \rightarrow A$$

This is the type of a function. The function infers a type A and a natural number n . It then takes a vector of length n where each element has type A and a natural number i such that i is less than n . It then returns an object of type A .



<https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2022p9q13.pdf>



(ii) Given the two following Agda declarations:

$$\begin{aligned} \text{zip} : & \forall \{A : \text{Set}\} B : \text{Set} n : \text{Nat} \rightarrow \\ & (\text{Vec } A \, n \times \text{Vec } B \, n) \rightarrow \text{Vec } (A \times B) \, n \end{aligned}$$
$$\begin{aligned} \text{unzip} : & \forall \{A : \text{Set}\} \{B : \text{Set}\} \{n : \text{Nat}\} \rightarrow \\ & \text{Vec } (A \times B) \, n \rightarrow \text{Vec } A \, n \times \text{Vec } B \, n \end{aligned}$$

Write a type expressing that `unzip` followed by `zip` is the identity.

$$\forall \{A : \text{Set}\} \{B : \text{Set}\} \{n : \text{Nat}\} \{v : \text{Vec } (A \times B) \, n\} \rightarrow (v \equiv \text{zip}(\text{unzip}(v)))$$



$$\begin{array}{c}
 \frac{f : (X \rightarrow Y) \rightarrow X, k : X \rightarrow Y \vdash f : (X \rightarrow Y) \rightarrow X, k : X \rightarrow Y \vdash k : (X \rightarrow Y)}{\frac{f : (X \rightarrow Y) \rightarrow X, k : X \rightarrow Y \vdash f : (X \rightarrow Y) \rightarrow X, k : X \rightarrow Y \vdash f(k) : X}{f : (X \rightarrow Y) \rightarrow X \vdash \text{letcont } k : \sim X. f(k) : X}} \\
 \frac{\cdot \vdash \lambda f : (X \rightarrow Y) \rightarrow X. \text{letcont } k : \sim X. f(k) : ((X \rightarrow Y) \rightarrow X) \rightarrow X}{\cdot \vdash \lambda f : (X \rightarrow Y) \rightarrow X. \text{letcont } k : \sim X. f(k) : ((X \rightarrow Y) \rightarrow X) \rightarrow X}
 \end{array}$$

Figure 4: A proof of Peirce's Law in the λ -calculus extended with continuations.



$$\frac{\frac{\Gamma; \Delta, C \vdash A \text{ true}}{\Gamma; \Delta, C \vdash A \text{ true}} \quad \frac{\frac{\Gamma; \Delta, C \vdash \text{true} \quad \Gamma; A; \Delta, C \vdash C \text{ false}}{\Gamma; A; \Delta, C \vdash \text{contr}}}{\Gamma; \Delta, C \vdash A \text{ false}} \quad \frac{\Gamma; \Delta, C \vdash \text{contr}}{\Gamma; \Delta \vdash C \text{ true}}$$

Figure 5: A proof that $\Gamma; \Delta \vdash A \text{ true} \wedge \Gamma, A; \Delta \vdash C \text{ true} \implies \Gamma; \Delta \vdash C \text{ true}$



$$\begin{array}{c}
 \frac{\frac{\frac{\Gamma, x : A \vdash A \text{ type}}{\Gamma, x : A \vdash \lambda y : A. \lambda p : (x = y : A). \text{subst}[z : A. (z = x : A)]. (p, \text{refl } x) : (y = x : A)}{\Gamma \vdash A \text{ type}} \quad \frac{\Delta \quad \nabla}{\Gamma \vdash \text{sym}_A : \Pi x : A. \Pi y : A. (x = y : A) \rightarrow (y = x : A)}}{\Gamma \vdash A \text{ type}} \\
 \\
 \Delta = \frac{\frac{\Gamma, x : A, y : A \vdash A \text{ type}}{\Gamma, x : A, y : A \vdash x : A} \quad \frac{\Gamma, x : A, y : A \vdash x : A}{\Gamma, x : A, y : A \vdash y : A}}{\Gamma, x : A, y : A \vdash (x = y : A) \text{ type}} \\
 \\
 \nabla = \frac{\frac{\Gamma' \vdash A \text{ type}}{\Gamma', z : A \vdash (z = x : A) \text{ type}} \quad \frac{\Gamma' \vdash z : A}{\Gamma' \vdash x : A} \quad \frac{\Gamma' \vdash p : (x = y : A)}{\Gamma \vdash \text{refl } x : [x/z](z = x : A)}}{\Gamma' \vdash \text{subst}[z : A. (z = x : A)]. (p, \text{refl } x) : (x = y : A) \rightarrow [y/v](v = x : A)}
 \end{array}$$



$$\begin{array}{c}
 \frac{\frac{\Gamma \vdash b_2 : [a_2/x]B}{\Gamma \vdash [a_2/x]B \text{ type}} \text{ Regularity}}{\Gamma \vdash (\text{subst}[x : A.B](p, b_1) = b_2 : [a_2/x]B) \text{ type}} \text{ Regularity} \\
 \frac{\frac{\Gamma \vdash b_2 : [a_2/x]B}{\Gamma \vdash [a_2/x]B \text{ type}} \text{ Regularity} \quad \frac{\Gamma, x : A \vdash B \text{ type} \quad \frac{\Gamma \vdash p : (a_1 = a_2 : A) \quad \Gamma \vdash b_1 : [a_1/x]B}{\Gamma \vdash b_2 : [a_2/x]B}}{\Gamma \vdash \text{subst}[x : A.B](p, b_1) : [a_2/x]B} \text{ Regularity}}{\Gamma \vdash (\text{subst}[x : A.B](p, b_1) = b_2 : [a_2/x]B) \text{ type}}
 \end{array}$$