# Types and Lambda Calculus Notes

by Tyler Wright

github.com/Fluxanoia fluxanoia.co.uk

*These notes are not necessarily correct, consistent, representative of the course as it stands today, or rigorous. Any result of the above is not the author's fault.*

**These notes are in progress.**

# Contents

# 1 Terms (1.1)

We suppose that we have a countably infinite set of variables $\mathbb{V}$ (we usually refer to elements of this set as $x$, $y$, $z$, etc.), from this we define the alphabet of lambda calculus $\mathbb{V} + \{\lambda, ., (, )\}$. The set of terms of lambda calculus $\Lambda$ is defined inductively for some $x$ in $\mathbb{V}$ by the variable axiom:

$$\frac{}{x \in \Lambda},$$

the application axiom:

$$\frac{M \in \Lambda \qquad N \in \Lambda}{(MN) \in \Lambda},$$

and the abstraction axiom:

$$\frac{M \in \Lambda}{(\lambda x.M) \in \Lambda}.$$

## 1.1 Subterms

Subterms of a term $M$ are substrings of $M$ that are themselves terms and not captured by a $\lambda$ (directly preceeded by).

## 1.2 Syntactical Conventions

Parentheses allow our lambda calculus to be unambigious, but for the sake of simplicity, we will construct conventions that will allow us to retain unique meaning with less parentheses:

- Omit outermost parentheses,
- Terms associate to the left, $(MNP)$ parses as $((MN)P)$,
- Bodies of abstractions end at parentheses, $(\lambda x.MN)$ parses as $(\lambda x.(MN))$,
- Group repeated abstractions, $(\lambda xy.M)$ parses as $(\lambda x.(\lambda y.M))$.

# 2 Alpha

## 2.1 Free Variables (2.1)

We define the function $FV : \Lambda \to \mathcal{P}(\mathbb{V})$, which returns the set of variables contained within a term $M$ that are not bound. We define it recursively on the structure of terms:

$$\begin{aligned} FV(x) &= \{x\}, \\ FV(MN) &= FV(M) \cup FV(N), \\ FV(\lambda x.M) &= FV(M) \setminus \{x\}. \end{aligned}$$

If a term has no free variables we say it is closed, and if a term has at least one free variable then we say it is open. The set of all closed terms is denoted by $\Lambda^0$.

## 2.2 Substitution (2.2)

We define 'capture-avoiding' substitution of a term $M$ for a variable $x$ recursively on the structure of terms:

$$\begin{array}{rcll} y[M/x] & = & y & \text{if } y \neq x, \\ y[M/x] & = & M & \text{if } y = x, \\ (PQ)[M/x] & = & P[M/x]Q[M/x], & \\ (\lambda y.P)[M/x] & = & \lambda y.P & \text{if } y = x, \\ (\lambda y.P)[M/x] & = & \lambda y.P[M/x] & \text{if } y \neq x \text{ and } y \notin FV(M). \end{array}$$

On the final case, we stipulate that $y$ cannot be a free variable of $M$ because otherwise free variables in the substitution could be captured by the lambda.

## 2.3 Alpha Equivalence (2.3)

Suppose we have a term $\lambda x.M$ and $y$ in $\mathbb{V} \setminus FV(M)$, then substituting $y$ for $x$ is a change of bound variable name. If two terms can be made identical through changes of bound variable name, they are $\alpha$-equivalent. The set of $\lambda$-terms is the set $\Lambda$ under $\alpha$-equivalence.

This equivalence is much more useful to us than string comparision, so for the remainder of the notes we will always be referring to $\lambda$-terms as terms.

## 2.4 The Variable Convention

For $M_1, \ldots, M_k$ terms occuring in the same scope, we assume each term has distinct bound variables. We can make this assumption as otherwise, we can use changes of bound variable names to make it so.

# 3 Beta (3.1-4)

The one-step $\beta$-reduction relation, denoted by $\rightarrow_\beta$, is inductively defined by the redex rule:

$$\frac{}{(\lambda x.M)N \rightarrow_\beta M[N/x]},$$

the left and right application rules:

$$\frac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N} \qquad \frac{N \rightarrow_\beta N'}{MN \rightarrow_\beta MN'},$$

and the abstraction rule:

$$\frac{M \rightarrow_\beta N}{\lambda x.M \rightarrow_\beta \lambda x.N}.$$

A term $M$ is said to be in $\beta$-normal form if there is no term $N$ such that $M \rightarrow_\beta N$. In general, $\beta$-reductions are sequences of consecutive one-step $\beta$-reductions:

$$M_0 \rightarrow_\beta M_1 \rightarrow_\beta \cdots \rightarrow_\beta M_k,$$

for some $k$ in $\mathbb{N}_0$. We say that $M_0$ $\beta$-reduces to $M_k$, denoted by $M_0 \twoheadrightarrow_\beta M_k$. If $M \twoheadrightarrow_\beta N$, we say that $N$ is a reduct of $M$, and is a proper reduct if $N \neq M$. If we can choose some $\beta$-normal $N$ such that $M \twoheadrightarrow_\beta N$ then $M$ is normalisable. If a term admits no infinite $\beta$-reductions then we say that it is strongly normalisable.

## 3.1 Standard Combinators

We have some interesting programs described below:

| | | | |
|---|---|---|---|
| $\mathbf{I}$ | $=$ | $\lambda x.x$ | $\mathbf{I}M \twoheadrightarrow_\beta M$ |
| $\mathbf{K}$ | $=$ | $\lambda xy.x$ | $\mathbf{K}MN \twoheadrightarrow_\beta M$ |
| $\mathbf{S}$ | $=$ | $\lambda xyz.xz(yz)$ | $\mathbf{S}MNP \twoheadrightarrow_\beta MP(NP)$ |
| $\omega$ | $=$ | $\lambda x.xx$ | $\omega M \twoheadrightarrow_\beta MM$ |
| $\Omega$ | $=$ | $\omega\omega$ | $\Omega \rightarrow_\beta \Omega$ |
| $\Theta$ | $=$ | $(\lambda xy.y(xxy))(\lambda xy.y(xxy))$ | $\Theta M \twoheadrightarrow_\beta M(\Theta M)$ |

## 3.2 Confluence of Beta (Thm. 3.1)

For a term $M$, if $M \twoheadrightarrow_\beta P$ and $M \twoheadrightarrow_\beta Q$ then there exists some term $N$ such that $P \twoheadrightarrow_\beta N$ and $Q \twoheadrightarrow_\beta N$.

## 3.3 Beta Convertibility (3.5)

For $M$ and $N$ terms, if $M$ and $N$ have a common reduct then we say that $M$ and $N$ are $\beta$-convertible, denoted by $M =_\beta N$.

# 4 Definability

## 4.1 Church Numerals (4.1)

The Church numeral for the number $n$ in $\mathbb{N}_0$, denoted by $\ulcorner n \urcorner$, is:

$$\lambda f x.\, \underbrace{f(\cdots(f}_{n \text{ times}} x)\cdots).$$

Each Church numeral is already in normal form.

## 4.2 Lambda Definability (4.2)

A function $f : \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ is $\lambda$-definable if there exists a $\lambda$-term $F$ that satifies:

$$F\ulcorner n_1 \urcorner \cdots \ulcorner n_k \urcorner =_\beta \ulcorner f(n_1, \ldots, n_k) \urcorner.$$

## 4.3 Basic Functions

We have addition, predecessor, subtraction, and the zero conditional defined here:

| | | | | |
|---|---|---|---|---|
| **Add** | $=$ | $\lambda yz.\lambda fx.yf(zfx)$ | $\mathbf{Add}\ulcorner m \urcorner\ulcorner n \urcorner =_\beta \ulcorner m+n \urcorner$ | |
| **Pred** | $=$ | $\lambda z.\lambda fx.z(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)$ | $\mathbf{Pred}\ulcorner 0 \urcorner =_\beta \ulcorner 0 \urcorner$<br>$\mathbf{Pred}\ulcorner n+1 \urcorner =_\beta \ulcorner n \urcorner$ | |
| **Sub** | $=$ | $\lambda mn.n\,\mathbf{Pred}\,m$ | $\mathbf{Sub}\ulcorner m \urcorner\ulcorner n \urcorner =_\beta \ulcorner 0 \urcorner$<br>$\mathbf{Sub}\ulcorner m \urcorner\ulcorner n \urcorner =_\beta \ulcorner m-n \urcorner$ | if $m-n<0$<br>otherwise |
| **IfZero** | $=$ | $\lambda\lambda xyz.x(\mathbf{K}\,z)y$ | $\mathbf{IfZero}\ulcorner 0 \urcorner\ulcorner p \urcorner\ulcorner q \urcorner =_\beta \ulcorner p \urcorner$<br>$\mathbf{IfZero}\ulcorner n+1 \urcorner\ulcorner p \urcorner\ulcorner q \urcorner =_\beta \ulcorner q \urcorner$ | |

## 4.4 Church Lists (4.3)

The Church encoding of a list $xs$ is the term $\ulcorner xs \urcorner$ defined recursively by:

$$\ulcorner \mathbf{Nil} \urcorner = \lambda cn.n,$$
$$\ulcorner x : xs \urcorner = \lambda cn.c\ulcorner x \urcorner(\ulcorner xs \urcorner cn).$$

This gives us the definition of **Cons**:

$$\mathbf{Cons} = \lambda xy.\lambda cn.cx(ycn).$$

We can treat this definition like a right fold, defining: $\mathbf{Sum} = \lambda x.x\,\mathbf{Add}\ulcorner 0 \urcorner$.

# 5 Recursion

## 5.1 Fixed Points (5.2)

We say that a term $N$ is a fixed point of another term $M$ if $MN =_\beta N$.

## 5.2 The First Recursion Theorem (Thm. 5.1)

Every term posesses a fixed point.

*Proof.* Let $M$ be a term. We define:

$$\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)),$$

called Curry's Paradoxical Combinator. We will show that $M(\mathbf{Y}M) =_\beta \mathbf{Y}M$ (so $\mathbf{Y}M$ is a fixed point of $M$):

$$\begin{aligned}\mathbf{Y}M &\to_\beta (\lambda x.M(xx))(\lambda x.M(xx)) \\ &\to_\beta M((\lambda x.M(xx))(\lambda x.M(xx)))\end{aligned}$$

$$M(\mathbf{Y}M) \to_\beta M((\lambda x.M(xx))(\lambda x.M(xx)))$$

$$M(\mathbf{Y}M) =_\beta \mathbf{Y}M,$$

as required. □

### 5.2.1 Solving Recursive Definitions

We want to solve for some term $M$:

$$\begin{aligned}Mx_1 \cdots x_n &=_\beta N[M/y] \\ &\iff M =_\beta \lambda x_1 \cdots x_n.N[M/y] \\ &\iff M =_\beta (\lambda y.\lambda x_1 \cdots x_n.N)M\end{aligned}$$

so if we find a fixed point of $(\lambda y.\lambda x_1 \cdots x_n.N)$ then we have an $M$ which satisfies the equation. But, by the First Recursion Theorem, we can always find such $M$. Thus, we can just set $M = \mathbf{Y}(\lambda y.\lambda x_1 \cdots x_n.N)$.

# 6 Induction

We let $\Phi$ be some property of terms. We have that if the following conditions are met then it follows that $\Phi$ holds for all terms:

1. For all variables $x$, $\Phi(x)$ holds,
2. For all terms $P$ and $Q$, if $\Phi(P)$ and $\Phi(Q)$ hold then $\Phi(PQ)$ holds,
3. For all terms $P$ and variables $x$, if $\Phi(P)$ holds then $\Phi(\lambda x.P)$ holds.

## 6.1 Free Variables of Substitutions of Unbound Variables (Lemma 6.1)

For all $\lambda$-terms $M$ and $N$, if $x$ is not in $FV(M)$ then $FV(M[N/x]) = FV(M)$. This can be proven by induction on the structure of $\lambda$-terms.

## 6.2 The Induction Metaprinciple (Prin. 6.1)

For some set $S$ with an inductive definition defined by rules $R_1, \ldots, R_k$. The induction principle for proving that for all $s$ in $S$, $\Phi(s)$ holds has $k$ clauses corresponding to the rules of $S$.

If a rule $R_i$ has $m$ premises and a side condition $\psi$:

$$\psi \, \frac{s_1 \in S \cdots s_m \in S}{s \in S}, \, (R_i)$$

then the corresponding clause in the induction principle requires that if $\Phi(s_1), \ldots, \Phi(s_m), \Phi(\psi)$ hold then $\Phi(s)$ holds.

# 7 Types (7.1)

We assume a countable set of type variables $\mathbb{A}$ (usually denoted by $a$, $b$, $c$, etc.). The monotypes $\mathbb{T}$ are a set of strings defined inductively by the type variable rule:

$$\frac{}{a \in \mathbb{T}},$$

for some $a \in \mathbb{A}$ and the arrow rule:

$$\frac{A \in \mathbb{T} \qquad B \in \mathbb{T}}{(A \to B) \in \mathbb{T}}.$$

We omit the outermost parentheses when writing these types, and they implicitly associate to the right.

Types schemes are pairs consisting of a finite set of type variables $a_1, \ldots, a_m$ and a monotype $A$ which we write as $\forall\, a_1 \cdots a_m.A$.

## 7.1 Free Type Variables (7.2)

We define the set of free type variables for a type scheme inductively with the following rules:

$$\begin{aligned} FTV(a) &= \{a\}, \\ FTV(A \to B) &= FTV(A) \cup FTV(B), \\ FTV(\forall\, a_1 \cdots a_m.A) &= FTV(A) \setminus \{a_1, \ldots, a_m\}. \end{aligned}$$

We consider type schemes that only differ by choice of bound variable names to be equivalent.

## 7.2 Type Substitution (7.3-4)

A type substitution is a total map $\sigma$ from $\mathbb{A}$ to $\mathbb{T}$ with the property that $\sigma(a) \neq a$ for only finitely many $a \in \mathbb{A}$. We define the map as follows:

$$\begin{aligned} a\sigma &= \sigma(a), \\ (A \to B)\sigma &= A\sigma \to B\sigma. \end{aligned}$$

### 7.2.1 Composition of Substitutions (7.5)

We write $\sigma_1\sigma_2$ for the substitution obtained by composing $\sigma_2$ after $\sigma_1$, defined as by $(\sigma_1\sigma_2)(a) = (\sigma_1(a))\sigma_2$.

# 8 The Type System

## 8.1 Type Assignment (8.1)

A type assignment is a pair of a term $M$ and a type scheme $A$ written $M : A$. The term part is called the subject and the type part the predicate.

## 8.2 Type Environments (8.2)

A type environment written $\Gamma$ is a finite set of type assignments of the form $x : \forall\, \bar{a}.A$ which are consistent in the sense that multiple type assignments of the same subject will agree. The subjects of $\Gamma$ is the set $\mathrm{dom}(\Gamma)$.

## 8.3 Type Judgement

A type judgement is a triple consisting of a type environment $\Gamma$, a term $M$, and a monotype $A$ written as $\Gamma \vdash M : A$.

## 8.4 The Type System (8.3)

The type system is defined by the following rules. For $x : \forall\, \bar{a}.A$ in $\Gamma$ we have the type variable rule:

$$\frac{}{\Gamma \vdash x : A[\bar{B}/\bar{a}]},$$

the rule of type application:

$$\frac{\Gamma \vdash M : B \to A \qquad \Gamma \vdash N : B}{\Gamma \vdash MN : A},$$

and for $x$ not in $\mathrm{dom}(\Gamma)$, we have the rule of type abstraction:

$$\frac{\Gamma \cup \{x : B\} \vdash M : A}{\Gamma \vdash \lambda x.M : B \to A}.$$

A proof tree justifying a type judgement is called a type derivation.

## 8.5 Typability (8.4, Lemma 8.1)

We say a closed term $M$ is typable if there is some type $A$ such that $\vdash M : A$. We have that $\lambda x.xx$ is untypable.

## 8.6 The Inversion Theorem (Thm. 8.1)

Suppose $\Gamma \vdash M : A$ is derivable, we have that:

- if $M$ is a variable $x$ then there is a type scheme $\forall\, \bar{a}.B$ in $\Gamma$ with $A = B[\bar{C}/\bar{a}]$ for some monotypes $\bar{C}$,
- if $M$ is an application $PQ$ then there is a type $B$ such that $\Gamma \vdash P : B \to A$ and $\Gamma \vdash Q : B$,
- if $M$ is an abstraction $\lambda x.P$ then there are types $B$ and $C$ such that $A = B \to C$ and $\Gamma \cup \{x : B\} \vdash P : C$.

## 8.7 Properties of the System (Lemma 9.1-2, Thm. 9.1-2)

For a term $M$, environment $\Gamma$ and type $S$, we have the following properties.

**Subterm Closure**
If $\Gamma \vdash M : S$ is derivable and $N$ is a subterm of $M$, there is some $\Gamma'$ containing $\Gamma$ and some $S'$ such that $\Gamma' \vdash N : S'$.

**Relevance ($1^{\mathbf{st}}$)**
If $\Gamma \vdash M : S$ then $FV(M) \subseteq \mathrm{dom}(\Gamma)$.

**Relevance ($2^{\mathbf{nd}}$)**
If $\Gamma \vdash M : S$ then $\{x : A$ where $x : A \in \Gamma$ and $x \in FV(M)\} \vdash M : S$.

**Weakening**
If $\Gamma \vdash M : S$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash M : S$.

**Subject Reduction**
If $\Gamma \vdash M : A$ and $M \to_\beta N$ then $\Gamma \vdash N : A$.

**Preservation under Substitution**
If $\Gamma \cup \{x : B\} \vdash M : A$ and $\Gamma \vdash N : B$ then $\Gamma \vdash M[N/x] : A$.

**Subformula**
We suppose $\Gamma$ has predicates solely consisting of monotypes, for some term $M$ in $\beta$-normal form with $\Gamma \vdash M : A$ we have that the derivation of this judgement is unique and all of the types mentioned in the derivation are substrings of types mentioned in the conclusion.

# 9 Type Constraints

A type constraint is a pair of monotypes $(A, B)$ written suggestively as $A \stackrel{?}{=} B$.

## 9.1 Constraint Generation

We create a constraint generating function, which takes in an environment and a term and returns a pair consisting of a type variable and a set of constraints:

$$
\begin{aligned}
&\mathrm{CGen}(\Gamma, x) = \\
&\quad \text{let } a \text{ be fresh} \\
&\quad \text{let } \forall\, a_1, \ldots, a_k.A = \Gamma(x) \\
&\quad \text{let } b_1, \ldots, b_k \text{ be fresh} \\
&\quad (a, \{a \stackrel{?}{=} A[b_1/a_1, \ldots, b_k/a_k]\})
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{CGen}(\Gamma, MN) = \\
&\quad \text{let } a \text{ be fresh} \\
&\quad \text{let } (b, \mathcal{C}_1) = \mathrm{CGen}(\Gamma, M) \\
&\quad \text{let } (c, \mathcal{C}_2) = \mathrm{CGen}(\Gamma, N) \\
&\quad (a, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{b \stackrel{?}{=} c \to a\})
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{CGen}(\Gamma, \lambda x.M) = \\
&\quad \text{let } a \text{ be fresh} \\
&\quad \text{let } b \text{ be fresh} \\
&\quad \text{let } (c, \mathcal{C}) = \mathrm{CGen}(\Gamma \cup \{x : b\}, M) \\
&\quad (a, \mathcal{C} \cup \{a \stackrel{?}{=} b \to c\})
\end{aligned}
$$

## 9.2 Unifier

A unifier to a finite set of type constraints $\{A_1 \stackrel{?}{=} B_1, \ldots, A_n \stackrel{?}{=} B_n\}$ is a type substitution $\sigma$ such that for all $i$ in $[n]$, $A_i\sigma = B_i\sigma$.

If $\mathrm{CGen}(\Gamma, M) = (a, \mathcal{C})$ then $\sigma$ is a unifier for $\mathcal{C}$ if and only if $\Gamma\sigma \vdash M : \sigma(a)$ is derivable.

### 9.2.1 Most General Unifier

For a set of type constraints $\mathcal{C}$, a unifier of $\mathcal{C}$, $\sigma$, is the most general unifier if every unifier $\sigma^*$ of $\mathcal{C}$ is of the form $\sigma\sigma'$ for some $\sigma'$.

## 9.3 Constraint Solved Form

A set of type constraints $\mathcal{C} = \{a_1 \stackrel{?}{=} A_1, \ldots, a_m \stackrel{?}{=} A_m\}$ is in solved form if each of the $a_i$ are distinct and do not occur in any $A_j$. If $\mathcal{C}$ is in solved form, we define $[[\mathcal{C}]] = [A_1/a_1, \ldots, A_m/a_m]$.

## 9.4 Robinson's Algorithm

Robinson'a algorithm is the repeated and exhaustive application of the following rules, returning the most general unifier from a set of constraints:

$$\begin{aligned}
\{A \stackrel{?}{=} A\} \uplus \mathcal{C} &\mapsto \mathcal{C}, \\
\{A_1 \to A_2 \stackrel{?}{=} B_1 \to B_2\} \uplus \mathcal{C} &\mapsto \{A_1 \stackrel{?}{=} B_1, A_2 \stackrel{?}{=} B_2\} \uplus \mathcal{C}, \\
\{A \stackrel{?}{=} a\} \uplus \mathcal{C} &\mapsto \{a \stackrel{?}{=} A\} \uplus \mathcal{C}, && (A \notin \mathbb{A}) \\
\{a \stackrel{?}{=} A\} \uplus \mathcal{C} &\mapsto \{a \stackrel{?}{=} A\} \uplus \mathcal{C}[A/a], && (a \notin FTV(A))
\end{aligned}$$

## 9.5 Hindley-Milner Type Inference

For an input, closed term $M$, we perform the following steps:

1. Generate the constraints $\mathcal{C}$ and type variable $a$ using $\text{CGen}(\varnothing, M)$,
2. Solve $\mathcal{C}$ using Robinson's algorithm to obtain the most general unifier or deduce insolvability,
3. If $\mathcal{C}$ has no solution, $M$ is untypable. Otherwise, we return $\sigma(a)$.

## 9.6 Principle Type Scheme Theorem

If a closed term $M$ is typable, then Hindley-Milner type inference returns a type $A$ that is principal meaning:

- $\vdash M : A$ is derivable,
- If $\vdash M : B$ is derivable, there exists monotypes $C_1, \ldots, C_k$ such that $B = A[C_1/a_1, \ldots, C_k/a_k]$.

# 10 Normalisation

## 10.1 Strong Normalisation

If $\Gamma \vdash M : A$ then $M$ is strongly normalising.

## 10.2 The Inhabitation Problem

The inhabitation problem is the problem of, given a type $A$, determining whether there is a closed term $M$ such that $\vdash M : A$.

## 10.3 Curry-Howard Theorem

For proofs in the implicational fragment of propositional logic, the following is true:

- From a constructive proof of $A$ from the initial assumptions $\Gamma$, we can extract a term $M$ such that $\Gamma \vdash M : A$,
- From each term $M$ with $\Gamma \vdash M : A$, we can extract a constructive proof that $A$ follows from the assumptions $\Gamma$.

## 10.4 Proof from Truth

A proof of $A$ from the starting assumptions $\Gamma$ is a certificate guaranteeing that $A$ is true if $\Gamma$ is true.

## 10.5 Truth from Proof

A proof of $A$ from the starting assumptions $\Gamma$ is a method of constructing evidence of $A$ from evidence of $\Gamma$.

## 10.6 The BHK Interpretation of Logic

The BHK interpretation of logic consists of the following rules:

- There can be no evidence for the truth from false,
- No evidence is needed for the truth of true,
- Evidence for $A$ and $B$ is a pair consisting of evidence for $A$ and evidence for $B$,
- Evidence for $A$ or $B$ is evidence of $A$ or evidence of $B$,

- Evidence of $A \Rightarrow B$ is a procedure for transforming evidence of $A$ to evidence of $B$,
- Evidence of $\neg A$ is a procedure for transforming evidence of $A$ to false,
- Evidence of $\forall\, x \in X, A$ is a procedure for transforming any $y$ and evidence of $y$ being in $X$ into evidence of $A[y/x]$,
- Evidence of $\exists\, x \in X, A$ is a triple consisting of a $y$, evidence of $y$ being in $X$, and evidence for $A[y/x]$.