Programming Language Theory

Primitive Recursion, General Recursion, and Polymorphism

陳亮廷 Chen, Liang-Ting
2020 邏輯、語言與計算暑期研習營
Formosan Summer School on Logic, Language, and Computation

Institute of Information Science, Academia Sinica

Gödel's T: Simply typed λ-calculus with naturals

The limit of $\lambda_\rightarrow$

Can you write this in $\lambda_\rightarrow$ using Church numerals?

$$\begin{aligned} \texttt{sum}(0) &= 0 \\ \texttt{sum}(1+n) &= (1+n) + f(n) \end{aligned}$$

It is not definable in $\lambda_\rightarrow$, since fixpoint operator is not allowed any more.

But, $\texttt{sum}$ is definable via *primitive recursion*: for some c and function g

$$\begin{aligned} \texttt{rec}(0, c, g(x,y)) &= c \\ \texttt{rec}(1+n, c, g(x,y)) &= g(n, \texttt{rec}(n, c, g(x,y))) \end{aligned}$$

$\lambda_\rightarrow$ with primitive recursion is called Gödel's T.

T: Types and terms

Definition 1 (Types)

$$\frac{B \in \mathbb{V}}{B : \texttt{Type}}\ (\text{tvar}) \qquad \frac{\sigma : \texttt{Type} \qquad \tau : \texttt{Type}}{\sigma \to \tau : \texttt{Type}}\ (\text{fun})$$

$$\frac{}{\mathbb{N} : \texttt{Type}}\ (\text{nat})$$

Definition 2 (Terms)

Additional term formation rules are added to $\lambda_{\to}$ as follows.

$$\frac{}{\texttt{zero} : \texttt{Term}_{\mathsf{T}}} \qquad \frac{M}{\texttt{suc}\ M : \texttt{Term}_{\mathsf{T}}}$$

$$\frac{L : \texttt{Term}_{\mathsf{T}} \qquad M : \texttt{Term}_{\mathsf{T}} \qquad N : \texttt{Term}_{\mathsf{T}} \qquad x \in V \qquad y \in V}{\texttt{rec}(M; x.\,y.\,N)\ L : \texttt{Term}_{\mathsf{T}}}$$

T: Typing rules

Definition 3

Additional term typing rules are added to $\lambda_{\to}$ as follows.

$$\frac{}{\Gamma \vdash \mathtt{zero} : \mathbb{N}} \qquad \frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \mathtt{suc}\ M : \mathbb{N}}$$

$$\frac{\Gamma \vdash L : \mathbb{N} \qquad \Gamma \vdash M : \tau \qquad \Gamma, x : \mathbb{N}, y : \tau \vdash N : \tau}{\Gamma \vdash \mathtt{rec}(M; x.\, y.\, N)\ L : \tau}$$

- Substitution for **T** is defined similarly.
- Substitution respects typing judgements, i.e. $\Gamma \vdash N : \tau$ and $\Gamma, x : \tau \vdash M : \sigma$, then $\Gamma \vdash M[N/x] : \sigma$.

T: Dynamics

β-conversion for **T** is extended with two rules

$$\texttt{rec}(M, x.\,y.\,N)\ \texttt{zero} \longrightarrow_\beta M$$
$$\texttt{rec}(M, x.\,y.\,N)\ \texttt{suc}\ L \longrightarrow_\beta N[L, \texttt{rec}(M; x.\,y.\,N)\ L/x, y]$$

Similarly, a β-reduction $\longrightarrow_{\beta 1}$ extends $\longrightarrow_\beta$ to all parts of a term and $\longrightarrow_{\beta *}$ indicates finitely many β-reductions.

Theorem 4

T enjoys the strong and weak normalisation properties as well as type safety.

Example: Addition and summation

$\texttt{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ can be defined in $\texttt{T}$ as

$$\lambda n.\, \lambda m.\, \texttt{rec}\ (m; x.\, y.\, \texttt{suc}\ y)\ n\ m$$

$\texttt{sum} : \mathbb{N} \to \mathbb{N}$ can be defined in $\texttt{T}$ as

$$\lambda n.\, \texttt{rec}\ (\texttt{zero}; x.\, y.\, \texttt{add}\ (\texttt{suc}\ x)\ y)\ n$$

Exercise

Evaluate $\texttt{sum}\ (\texttt{suc zero})$.

PCF— System of Recursive Functions

PCF: $\lambda_{\rightarrow}$ with naturals and general recursion

T does not include all computable functions, since all terms terminate eventually. Programming language in reality allows us to do *general recursion* including *infinite loops*.

What to do if we want type and general recursion at the same time?

PCF: Types and terms

Definition 5 (Types)

PCF has the same class of types as **T**.

Definition 6 (Terms)

Additional term formation rules are added to $\lambda_{\rightarrow}$ as follows.

$$\frac{}{\texttt{zero} : \texttt{Term}_{\mathsf{PCF}}} \qquad \frac{M : \texttt{Term}_{\mathsf{PCF}}}{\texttt{suc}\ M : \texttt{Term}_{\mathsf{PCF}}}$$

$$\frac{L : \texttt{Term}_{\mathsf{PCF}} \qquad M : \texttt{Term}_{\mathsf{PCF}} \qquad N : \texttt{Term}_{\mathsf{PCF}} \qquad x \in V}{\texttt{ifz}(M; x.\, N)\ L}$$

$$\frac{M : \texttt{Term}_{\mathsf{PCF}} \qquad x \in V}{\texttt{fix}\, x.\, M : \texttt{Term}_{\mathsf{PCF}}}$$

PCF: Typing rules

Definition 7

Additional term typing rules are added to $\lambda_{\to}$ as follows.

$$\frac{}{\Gamma \vdash \texttt{zero} : \mathbb{N}} \qquad \frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \texttt{suc}\, M : \mathbb{N}}$$

$$\frac{\Gamma \vdash L : \mathbb{N} \qquad \Gamma \vdash M : \tau \qquad \Gamma, x : \mathbb{N} \vdash N : \tau}{\Gamma \vdash \texttt{ifz}(M; x.\, N)\; L : \tau}$$

$$\frac{\Gamma, x : \tau \vdash M : \tau}{\Gamma \vdash \texttt{fix}\, x.\, M : \tau}$$

- Substitution for **PCF** is defined similarly.
- Substitution respects typing judgements, i.e. $\Gamma \vdash N : \tau$ and $\Gamma, x : \tau \vdash M : \sigma$, then $\Gamma \vdash M[N/x] : \sigma$.

PCF: Dynamics

β-conversion for **PCF** is extended with three rules

$$\begin{aligned} \texttt{fix}\, x.\, M &\longrightarrow_\beta M[\texttt{fix}\, x.\, M/x] \\ \texttt{ifz}(M; x.\, N)\ \texttt{zero} &\longrightarrow_\beta M \\ \texttt{ifz}(M; x.\, N)\ (\texttt{suc} M) &\longrightarrow_\beta N[M/x] \end{aligned}$$

Similarly, a β-reduction $\longrightarrow_{\beta 1}$ extends $\longrightarrow_\beta$ to all parts of a term and $\longrightarrow_{\beta *}$ indicates finitely many β-reductions.

Theorem 8

PCF enjoys type safety.

Example

A term which never terminates can be defined easily.

$$
\begin{aligned}
& \texttt{fix}\, x.\, x && \longrightarrow_{\beta 1} x[\texttt{fix}\, x.\, x/x] \\
\equiv\ & \texttt{fix}\, x.\, x && \longrightarrow_{\beta 1} x[\texttt{fix}\, x.\, x/x] \\
\equiv\ & \texttt{fix}\, x.\, x && \longrightarrow_{\beta 1} x[\texttt{fix}\, x.\, x/x] \\
\equiv\ & \ldots
\end{aligned}
$$

Example: Predecessor and negation

$$\texttt{pred} := \lambda n : \mathbb{N}.\, \texttt{ifz}(\texttt{zero}; x.\, x)\; n \qquad : \mathbb{N} \to \mathbb{N}$$

$$\texttt{not} := \lambda n : \mathbb{N}.\, \texttt{ifz}(\texttt{suc zero}; x.\, \texttt{zero})\; n \qquad : \mathbb{N} \to \mathbb{N}$$

Exercise

Evaluate the following terms to their normal forms.

1. `pred zero`
2. `pred (suc suc suc zero)`
3. `not (suc suc zero)`

F — Polymorphic Typed λ-Calculus

Polymorphic types

Given type variables $\mathbb{V}$, τ : Type is defined by defined by

$$\frac{t \in \mathbb{V}}{t : \mathsf{Type}} \text{ (tvar)}$$

$$\frac{\sigma : \mathsf{Type} \qquad \tau : \mathsf{Type}}{\sigma \to \tau : \mathsf{Type}} \text{ (fun)}$$

$$\frac{\sigma : \mathsf{Type} \qquad t \in \mathbb{V}}{\forall t.\, \sigma : \mathsf{Type}} \text{ (poly)}$$

where t may or may not appear in σ.

The polymorphic type $\forall t.\, \sigma$ provides a generic type for every instance $\sigma[\tau/t]$ whenever t is instantiated by an actual type τ.

Examples

- $\texttt{id} : \forall t.\, t \to t$
- $\texttt{proj}_1 : \forall t.\, \forall u.\, t \to u \to t$
- $\texttt{proj}_2 : \forall t.\, \forall u.\, t \to u \to u$
- $\texttt{length} : \forall t.\, \texttt{list}\ t \to \texttt{nat}$
- $\texttt{singleton} : \forall t. t \to \texttt{list}(t)$

Free and bound variables, again

Definition 9

The *free variable* $\mathsf{FV}(\tau)$ of τ is defined inductively by

$$\begin{aligned} \mathsf{FV}(t) &= t \\ \mathsf{FV}(\sigma \to \tau) &= \mathsf{FV}(\sigma) \cup \mathsf{FV}(\tau) \\ \mathsf{FV}(\forall t.\, \sigma) &= \mathsf{FV}(\sigma) - \{t\} \end{aligned}$$

For convenience, the function extends to contexts:

$$\mathsf{FV}(\Gamma) = \{\, t \in \mathbb{V} \mid \exists (x : \sigma) \in \Gamma \wedge t \in \mathsf{FV}(\sigma) \,\}.$$

1. $\mathsf{FV}(t_1) = \{t_1\}$.
2. $\mathsf{FV}(\forall t.\, (t \to t) \to t \to t) = \emptyset$.
3. $\mathsf{FV}(x : t_1, y : t_2, z : \forall t.\, t) = \{t_1, t_2\}$.

Capture-avoiding substitution for type

Definition 10

The *(capture-avoidance) substitution* of a type ρ for the free occurrence of a type variable t is defined by

$$
\begin{aligned}
t[\rho/t] &= \rho \\
u[\rho/t] &= u && \text{if } u \neq t \\
(\sigma \to \tau)[\rho/t] &= \sigma[\rho/t] \to \tau[\rho/t] \\
(\forall t.\sigma)[\rho/t] &= \forall t.\sigma \\
(\forall u.\sigma)[\rho/t] &= \forall u.\sigma[\rho/t] && \text{if } u \neq t, u \notin \mathsf{FV}(\rho)
\end{aligned}
$$

Recall that $u \notin \mathsf{FV}(\rho)$ means that u is *fresh* for ρ.

Typed terms

Definition 11

On top of $\lambda_{\rightarrow}$, $\mathtt{F}$ has additional term formation rules as follows.

$$\frac{M : \mathtt{Term}_F \qquad t : \mathbb{V}}{\Lambda t.\, M : \mathtt{Term}_F} \text{(gen)}$$

$$\frac{M : \mathtt{Term}_F \qquad \tau : \mathtt{Type}}{M\ \tau : \mathtt{Term}_F} \text{(inst)}$$

1. $\Lambda t.\, M$ for type abstraction, or *generalisation*.
2. $M\ \tau$ for type application, or *instantiation*.

Example

Suppose $\texttt{length} : \forall t.\, \texttt{list}\ t \to \texttt{nat}$.

Then,

1. $\texttt{length}\ \texttt{nat}$
2. $\texttt{length}\ \texttt{bool}$
3. $\texttt{length}\ (\texttt{nat} \to \texttt{nat})$

are instances of $\texttt{length}$ with types

1. $\texttt{list}\ \texttt{nat} \to \texttt{nat}$
2. $\texttt{list}\ \texttt{bool} \to \texttt{nat}$
3. $\texttt{list}\ (\texttt{nat} \to \texttt{nat}) \to \texttt{nat}$

System F: Typing judgement

A *type context* is a sequence of pairs of type variable and a type

$$t : \tau$$

F has two kinds of typing judgements.

- $\Delta \vdash \tau$ for τ for a valid type under the type context Δ
- $\Delta; \Gamma \vdash M : \tau$ for a well-typed term under the context Γ and the type context Δ.

For example,

$$t : \tau_1 \vdash t \to t$$

is a judgement saying that $t \to$ is a valid type under the type context $(t : \tau_1)$.

Then, we have to *justify* why this judgement holds.

System F: Type formation

The justification of $\Delta \vdash \tau$ is constructed inductively by following rules.

$$\frac{t \in \Delta}{\Delta \vdash t} \qquad \frac{\Delta, t \vdash \tau}{\Delta \vdash \forall t.\, \tau}$$

$$\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}$$

Exercise

Derive the judgement

$$t : \tau \vdash t \to t$$

System F: Typing rules

The justification of $\Delta; \Gamma \vdash M : \sigma$ is defined inductively by following rules.

$$\frac{x : \sigma \in \Gamma}{\Delta; \Gamma \vdash x : \sigma} \qquad \frac{\Delta, t; \Gamma \vdash M : \sigma}{\Delta; \Gamma \vdash \Lambda t.\, M : \forall t.\, \sigma} \ (\forall\text{-intro})$$

$$\frac{\Delta; \Gamma \vdash M : \sigma \to \tau \qquad \Delta; \Gamma \vdash N : \sigma}{\Delta; \Gamma \vdash M\ N : \tau}$$

$$\frac{\Delta \vdash \sigma \qquad \Delta; \Gamma, x : \sigma \vdash M : \tau}{\Delta; \Gamma \vdash \lambda x : \sigma.\, M : \sigma \to \tau} \qquad \frac{\Delta; \Gamma \vdash M : \forall t.\, \sigma \qquad \Delta \vdash \tau}{\Delta; \Gamma \vdash M\ \tau : \sigma[\tau/t]} \ (\forall\text{-elim})$$

For convenience, $\vdash M : \tau$ stands for $\cdot; \cdot \vdash M : \tau$.

Typing derivation

The typing judgement $\vdash \Lambda t.\, \Lambda u.\, \lambda(x : t)(y : u).\, x : \forall t.\ t \to u \to t$ is derivable from the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{t \in t, u}{t, u \vdash t} \qquad \dfrac{\dfrac{u \in t, u}{t, u \vdash u} \qquad \dfrac{x : t \in (x : t, y : u)}{t, u; x : t, y : u \vdash x : t}}{t, u; x : t \vdash \lambda(y : u).\ x : u \to t}}{t, u; \cdot \vdash \lambda(x : t)(y : u).\ x : t \to u \to t}}{t; \cdot \vdash \Lambda u.\, \lambda(x : t)(y : u).\ x : \forall u.\, t \to u \to t}}{\vdash \Lambda t.\, \Lambda u.\, \lambda(x : t)(y : u).\ x : \forall t.\, \forall u.\, t \to u \to t}$$

Exercise

Derive the following judgements:

1. $\vdash \Lambda t.\, \lambda(x : t).\, x : \forall t.\ t \to t$
2. $\sigma; a : \sigma \vdash (\Lambda t.\, \lambda(x : t)(y : t).\, x)\ \sigma\ a : \sigma \to \sigma$
3. $\vdash \Lambda t.\, \lambda(f : t \to t)(x : t).\, f\,(f\,x) : \forall t.\ (t \to t) \to t \to t$

Hint. **F** is syntax-directed, so the type inversion holds.

System F: β-reduction

The β-conversion has two rules

$$(\lambda(x:\tau).\,M)\,N \longrightarrow_\beta M[x/N] \quad \text{and} \quad (\Lambda t.\,M)\;\tau \longrightarrow_\beta M[\tau/t]$$

For example,

$$(\Lambda t.\lambda x:t.\,x)\;\tau\;a \longrightarrow_\beta (\lambda x:t.\,x)[\tau/t]\;a \equiv (\lambda x:\tau.\,x)\;a \longrightarrow_\beta x[a/x] \equiv a$$

Similarly, β-conversion extends to subterms of a given term, introducing symbols $\longrightarrow_{\beta 1}$ and $\longrightarrow_{\beta *}$ in the same way.

Self application

Self-application is not typable in simply typed λ-calculus.

$$\lambda(x : t).\, x\, x$$

However, self-application is possible in System F.

$$\lambda(x : \forall t.t \to t).\, x\ (\forall t.t \to t)\ x$$

Exercise

Instantiate the first t with the type $\forall t.\, t \to t$.

Sum type

Definition 12

The *sum type* is defined by

$$\sigma + \tau := \forall t.(\sigma \to t) \to (\tau \to t) \to t$$

It has two injection functions: the first injection is defined by

$$\texttt{left}_{\sigma+\tau} := \lambda(x : \sigma).\, \Lambda t.\, \lambda(f : \sigma \to t)(g : \tau \to t).\, f\, x$$

$$\texttt{right}_{\sigma+\tau} := \lambda(y : \tau).\, \Lambda t.\, \lambda(f : \sigma \to t)(g : \tau \to t).\, g\, y$$

Exercise

Define

$$\texttt{either} : \forall u.\, (\sigma \to u) \to (\tau \to u) \to (\sigma + \tau \to u) \to u$$

Product type

Definition 13 (Product Type)

The product type is defined by

$$\sigma \times \tau := \forall t.(\sigma \to \tau \to t) \to t$$

The pairing function is defined by

$$\langle \_, \_ \rangle := \lambda(x : \sigma)(y : \tau).\, \Lambda t.\, \lambda(f : \sigma \to \tau \to t).\, f\, x\, y$$

Exercise

Define projections

$$\mathtt{proj}_1 : \sigma \times \tau \to \sigma \quad \text{and} \quad \mathtt{proj}_2 : \sigma \times \tau \to \tau$$

Natural Numbers i

The type of Church numerals is defined by

$$\texttt{nat} := \forall t.\,(t \to t) \to t \to t$$

Church numerals

$$\begin{aligned} \mathsf{c}_n &: \texttt{nat} \\ \mathsf{c}_n &:= \Lambda t.\,\lambda(f : t \to t)\,(x : t).\, f^n\,x \end{aligned}$$

Successor

$$\texttt{suc} : \texttt{nat} \to \texttt{nat}$$
$$\texttt{suc} := \lambda(n : \texttt{nat}).\, \Lambda t.\, \lambda(f : t \to t)\,(x : t)\,.\, f\,(n\; t\; f\; x)$$

Addition

$$\texttt{add} : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$$
$$\texttt{add} := \lambda(n : \texttt{nat})\,(m : \texttt{nat}) \quad \Lambda t.\, \lambda(f : t \to t)\,(x : t).\; (m\; t\; f)\;(n\; t\; f\; x)$$

Natural Numbers iii

Multiplication

$$\texttt{mul} : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$$
$$\texttt{mul} := ?$$

Conditional

$$\texttt{ifz} : \forall t.\, \texttt{nat} \to t \to t \to t$$
$$\texttt{ifz} := ?$$

System F allows us to define *iterator* like `fold` in Haskell.

$$\texttt{fold}_{\texttt{nat}} : \forall t.\, (t \to t) \to t \to \texttt{nat} \to t$$

$$\texttt{fold}_{\texttt{nat}} := \Lambda t.\, \lambda(f : t \to t)(e_0 : t)(n : \texttt{nat}).n\ t\ f\ e_0$$

Exercise

Define `add` and `mul` using $\texttt{fold}_{\texttt{nat}}$ and justify your answer.

1. $\texttt{add}' := ? : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$
2. $\texttt{mul}' := ? : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$

Lists

Definition 14

For any type σ, the type of lists over σ is

$$\texttt{list}\,\sigma := \forall t.\, t \to (\sigma \to t \to t) \to t$$

with "list constructors":

$$\texttt{nil}_\sigma := \Lambda t.\lambda(h : t)(f : \sigma \to t \to t).\, h$$

and

$$\texttt{cons}_\sigma := \lambda(x : \sigma)(xs : \texttt{list}\,\sigma).\, \Lambda t.\lambda(h : t)(f : \sigma \to t \to t).f\,x\,(xs\ t\ h\ f)$$

of type $\sigma \to \texttt{list}\,\sigma \to \texttt{list}\,\sigma$.

Type safety and normalisation

Theorem 15 (Type safety)

Suppose $\vdash M : \sigma$. *Then,*

1. $M \longrightarrow_{\beta 1} N$ *implies* $\vdash N : \sigma$;
2. *M is in normal form or there exists N such that* $M \longrightarrow_{\beta 1} N$

Type safety is proved by induction on the derivation of $\vdash M : \sigma$.

Theorem 16 (Normalisation properties)

F enjoys the weak and strong normalisation properties.

Proved by Girard's *reducibility candidates*.

Type erasure

Definition 17

The *erasing map* is a function defined by

$$
\begin{aligned}
|x| &= x \\
|\lambda(x : \tau).\, M| &= \lambda x.\, |M| \\
|M\ N| &= (|M|\ |N|) \\
|\Lambda t.\, M| &= |M| \\
|M\ \tau| &= |M|
\end{aligned}
$$

Proposition 18

Within System F, if $\vdash M : \sigma$ *and* $|M| \longrightarrow_{\beta 1} N'$*, then there exists a well-typed term* N *with* $\vdash N : \sigma$ *and* $|N| = N'$.

Undecidability of type inference

Theorem 19

It is undecidable whether, given a closed term M of the untyped lambda-calculus, there is a well-typed term M' in System F such that $|M'| = M$.

Arbitrary Rank Polymorphism $\forall$ can appear anywhere (GHC with `-XRankNType`).

Rank-1 Polymorphism $\forall$ only appear in the outermost position.

Hindley-Milner type system adapted by Haskell 98, Standard ML, etc. supports only rank-1 polymorphism, so type inference is still decidable.

Parametricity

What functions can you write for the following type?

$$\forall t.\, t \to t$$

Since t is arbitrary, we cannot inspect the content of t. What we can do with t is simply return it.

Theorem 20

Every term M of type $\forall t.\, t \to t$ is observationally equivalent[1] to $\Lambda t.\, \lambda x : t.\, x$.

[1] The notion of observational equivalence is beyond the scope of this lecture.

Parametricity: Theorems for free[2]

Assume F extended with the list type $\texttt{list}\ \tau$ for τ and the type $\mathbb{N}$ of naturals, denoted $\mathsf{F}_{\texttt{list},\mathbb{N}}$.

Then $\texttt{head} \circ \texttt{map}\ f = f \circ \texttt{head}$ for any $f : \tau \to \sigma$ where $\texttt{head} : \forall t.\ \texttt{list}\ t \to t$ can be proved by just reading the type of `head` and `tail`!

Theorem 21

For any type σ in F (with lists) and $\cdot \vdash M : \sigma$, then

$$M \sim M : \mathcal{R}_{\sigma,\sigma}$$

[2] Philip Wadler. 1989. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA '89).* ACM, New York, NY, USA, 347–359.

Homework

1. (25%) Extend **PCF** with the type $\mathbb{B}$ of boolean values with $\texttt{ifz}(M; N)\ \texttt{true} =_\beta M$ and $\texttt{ifz}(M; N)\ \texttt{false} =_\beta N$ including term formation rules, typing rules, and dynamics for $\mathbb{B}$.
2. (25%) Define **pred** in **T** such that $\texttt{pred zero} = \texttt{zero}$ and $\texttt{pred}\ (\texttt{suc}\ n) = n$.
3. (25%) Define **even** in **PCF** such that **even** $n = \texttt{suc zero}$ if n is an even number; **even** $n = \texttt{zero}$ otherwise.
4. (25%) Define $\texttt{length}_\sigma : \texttt{list}\ \sigma \to \texttt{nat}$ calculating the length of a list.
5. (0%) Read the paper by Wadler (1989).