第 XI 部

Haskell 再考

Kinokkory (@shiatsumat)

まえがき

僕はこの記事を書いたのは，数学的論証が関数型プログラミングについてよりよい直感を与えてくれるということを伝えたいからです．根っからの C++ 好きであった僕は，縁あって Haskell を学んでいくうちに，その奥深い魅力を知りました．それまでにも色々なプログラミングを学んで来たつもりでしたが，Haskell を学ぶうちにプログラミングに対する考え方が大きく変わりました．数学由来の抽象的な概念が，Haskell のプログラムの至る所で活きているのです．この記事では数学的な性質をゆっくりとたどっていきます．遠回りに見えるかもしれませんが，深い理解が真の応用につながると僕は信じています．Haskell に馴染みがある人もそうでない人も，ぜひ読んでみてください．きっと何かしらの発見があると信じています．Haskell を通して得られるたくさんの知恵は，Haskell を使わなくても必ず役に立つはずです．「学問に王道なし」「下学して上達す」「すぐ役に立つことは，すぐに役立たなくなる」これらの言葉の意味はここに書くまでもないでしょう．[1] 恐れることはありません．Haskell の世界に真正面からぶつかっていってください．必ず大きなものが得られます．初学者の書いた拙い文章ですが，思いが伝わったら嬉しいかぎりです．もし誤りや意見があれば気兼ねなく webmaster@npca.jp まで連絡してください．ではでは，気楽にのんびり読んでいってください！

[1] 僕自身，心当たりが多いですが……

目次

第一章

はじめに

1　Haskell はどんな言語か

HaskellWiki http://www.haskell.org/haskellwiki/Haskell は Haskell を次のように紹介しています（拙訳）.

> Haskell は先進的な純粋関数型言語です．20 年以上の最前線の研究が生んだオープンソースな成果物により，堅牢で簡潔で正確なソフトウェアを作ることができます．多言語との統合の強力な支援，組み込みの並行処理・並列処理，デバッガ，プロファイラ，豊富なライブラリ，そして活発なコミュニティがある Haskell を使えば，柔軟で維持可能で高品質なソフトウェアをもっと簡単に作り出すことができます．

これ以上ここで書く必要はないでしょう．

2　道案内

Haskell に関する情報が一番まとまっているのは HaskellWiki http://www.haskell.org/haskellwiki/Haskell です．ここをしっかり読めばいいと思います．英語文献を含めれば情報は十分にありますが，とりあえずここでいろいろ掻い摘んで説明しておきます．

開発環境は Windows, Mac, Linux のどの OS でも最新版の Haskell Platform を使うのがお勧めです．コンパイラである Glasgow Haskell Compiler (GHC) に加えてさまざまツールやライブラリの安定版が入っています．まだ Haskell Platform をインストールしていないならば http://www.haskell.org/platform/ にアクセスしましょう．この記事のコードは Haskell Platform 2013.2.0.0 だけで動きます．

GHC のユーザーズガイドは http://www.haskell.org/ghc/docs/latest/html/users_guide/，その邦訳は http://www.kotha.net/ghcguide_ja/latest/ にあるので活用してください．ツールの使い方に加えて言語機能に関する詳細な説明があります．

Haskell のきちんとした言語仕様については http://www.haskell.org/haskellwiki/Language_and_library_specification を見てください．

ライブラリのインストールには cabal が使えます．Haskell Platform にはもとから入っています．Haskell ではライブラリを「パッケージ」として扱うのが一般的で，cabal はパッケージを作ったり管理したりインストールしたりするためのシステムです．Hackage http://hackage.haskell.org/ には

cabal によって作られた様々なパッケージが保管されています．

cabal にはいろいろな機能がありますが，とりあえず Hackage の parsec というパッケージをインストールしたい場合，

```
cabal install parsec
```

とすればいいです．

cabal の情報は http://www.haskell.org/cabal/ にまとまっています．

Haskell の API の検索には Hoogle[1] http://www.haskell.org/hoogle/ や Hayoo! http://holumbus.fh-wedel.de/hayoo/hayoo.html が使えます．これらのページはおそらく Haskell を使っていて最も見るページですから，ブックマークに登録するなり，ブラウザの検索バーに登録するなりしておくことをお勧めします．

日本語の書籍としては以下のものがあります．

- 『プログラミング Haskell』[4] — Haskell を基礎から丁寧に書いてある．
- 『すごい Haskell たのしく学ぼう！』[6] — めっちゃ楽しくて頭がヤバくてハイテンション．入門書だけど内容は結構高度．
- 『Real World Haskell — 実戦で学ぶ関数型言語プログラミング』[9] — 重たい．実世界の重さ．
- 『関数プログラミング入門 — Haskell で学ぶ原理と技法』[1] — 純粋静的型付け関数プログラミングを Haskell を通じて学べる．数学的側面が強い．普通じゃないプログラマのための Haskell 入門．
- 『関数プログラミングの楽しみ』[3] — 一流の人々による一流の本．めっちゃ面白い．一歩上を行く Haskell プログラマのための本．

英語の本については http://www.haskell.org/haskellwiki/Books を見てください．特に『Parallel and Concurrent Programming in Haskell』[7] は良書です（じきに邦訳が出るでしょう）．

Haskell からはすこし離れますが，この本もおすすめです．

- 『型システム入門 — プログラミング言語と型の理論』[10] — 型理論についての日本で初めての総合的教科書．非常に丁寧で，さまざまな世界が見えてくる．原題は『Types and Programming Languages』で，通称「TaPL（タプル）」．続編の『Advanced Topics in Types and Programming Languages』[11] も面白い．

ネット上のまとまった文献としては以下のものがあります．

- 『Typeclassopedia』http://www.haskell.org/haskellwiki/Typeclassopedia 邦訳 http://snak.tdiary.net/20091020.html — 型クラスを真正面から語っている．
- 『本物のプログラマは Haskell を使う』http://itpro.nikkeibp.co.jp/article/COLUMN/20060915/248215/ — 実践的で新しい話題も多くて面白い．
- 『Haskell - Wikibooks』http://en.wikibooks.org/wiki/Haskell 日本語版 http://ja.wikibooks.org/wiki/Haskell — これがウィキブックスだ．

[1] Foogle http://www.foogle.co.nr/ とは関係ありません．

- 『What I Wish I Knew When Learning Haskell』http://dev.stephendiehl.com/hask/ — Haskell について，入門書より少し先の話題を分かりやすく概説している．
- 『Okmij.org』http://okmij.org/ftp/ — 関数型言語に関する情報を深く幅広く扱っている．

その他のチュートリアルについては http://www.haskell.org/haskellwiki/Tutorials を見てください．

ネットに公開されている英語論文にも面白いものがかなりあるので，ぜひいろいろ読んでみてください．Haskell の高度な話題のほとんどは，本やブログではなく論文に書いてあります．まずは http://www.haskell.org/haskellwiki/Research_papers を眺めてみてください．

Haskell のコーディングのガイドラインとしては

- Haskell Style Guide https://github.com/tibbe/haskell-style-guide/blob/master/haskell-style.md
- Programming guidelines http://www.haskell.org/haskellwiki/Programming_guidelines

が参考になるでしょう．実際問題，統一されたルールは無いのですが，迷った時にはガイドラインを見るのも手です．

Haskell を使っている人は世界中にいます．Freenode IRC network にある #haskell というチャンネルは質問をするのにいい場所です．オープンなコミュニティで，親切で優秀な人がたくさんいますから，臆せず質問してみたらいいと思います．とりあえず http://ircbrowse.net/haskell にアクセスしてみてください．

また，きちんとした答えが欲しければ Stack Overflow http://stackoverflow.com/ で質問するのもいいでしょう．

3 この記事の読み方

この記事を読むにあたっては，基本的な Haskell の読み書き能力があるほうがいいと思います．でも，Haskell についての高度な前提知識は特に要らないと思います．分からないことがあれば，前の節で挙げた文献も必要に応じて参照してください．

この記事のこれ以降の雰囲気を説明しておきます．第二章は手馴らしです．第三章は途中から抽象的な議論になって少し難しくなりますが，非常に基本的なことが書いてあります．第四章と第五章はおそらく最も読みやすいです．とりあえずここから読み始めてみるという読み方もいいかと思います．とにかく好きなように読んでくださったら結構です．

この記事では少しばかり数学的な議論が出てきますが，高校数学程度の知識で読めるはずです．「$\because \cdots$」が「なぜならば…」，「$\therefore \cdots$」が「ゆえに…」，「$X \triangleq \cdots$」が「X は…として定義されている」という意味であることくらいを分かっておけばいいでしょう．だいたいの記号の定義はきちんと文章中に書いてあります．必要ならば索引も参照してください．

この記事は実装を出来るかぎり明示する方針で書いています．既存のライブラリの内部実装と違う場合もあるので，気を付けてください．この記事のソースコードを手で写して実行する場合は，Prelude との衝突を避けて

```
import Prelude ()
```

などとするのをお勧めします．また，GHC の拡張機能を使うべき場面も多いでしょう．とりあえず，

```
{–# LANGUAGE ExistentialQuantification, RankNTypes,
             EmptyDataDecls, RecursiveDo, TypeOperators, Arrows #–}
```

というプラグマをソースコードの冒頭に付ければ大丈夫だと思います．

Haskell の各種演算子については，見易さのために ASCII 文字ではなく記号を用いています．普通の記号があるものについて対応関係を列挙すると，

(*) = (×), (.) = (○), (<$>) = (⟨$⟩), (<*>) = (⊛), (>>>) = (⋙),
(>>=) = (▷), (>>) = (≫), (>=>) = (♢)

となっています．索引にも記号をきちんと並べてあるので，必要ならば参照してください．

第二章

基本事項

1　値

Haskell は**参照透明性** (referential transparency) の保たれている稀有なプログラミング言語です．多くのプログラミング言語では，プログラムの挙動を考える際に，評価の順番を考えたり，変数の値の変動を辿ったりしなければいけませんが，Haskell においては（IO モナドなどを考えなければ）プログラムは単純に式（あるいは項）であり，式を簡約していった結果の「**値**」さえ見ればプログラムの挙動は分かります．たとえば，プログラム中に gcd 36 24 という部分があれば，単純に 12 に置き換えてしまってもプログラムの挙動はほぼ変わりません．Haskell でプログラミングをしていれば，日常的にこの感覚を得ると思います．

しかし，話が複雑になってきたり，一般的・抽象的になってきたりすると，プログラムの姿は見えづらくなってきます．特に，無限リストを扱うとき，特に fibs $=$ 0 : 1 : zipWith (+) fibs (tail fibs) のような奇妙な再帰を見たとき，どこから手を付ければいいのか分からないような気持ちになるでしょう．

しかし，基本に忠実になって順を追って考えていけば，プログラムの挙動はおのずと見えてきます．Haskell は非常にシンプルな言語ですから，必要な道具は意外と少ないのです．

少しインフォーマルに「値」という言葉を導入しましたが，より専門的には「**表示** (denotation)」といいます．そして，式を「表示」として適切に解釈するための理論を**表示的意味論** (denotational semantics) といいます．名前がやたら強そうですが，臆することはありません．なお，この章ではかなり直感的な形で「値」あるいは「表示」を扱いますが，第三章においてもう少しきちんと表示的意味論を扱います．

式を「表示」として実際に数学的対象に対応付けることもできますが，そうすると Haskell の世界と数学の世界を行き来しなければならず大変ですから，ここでは「式の等しさ」のみを考えることにします．この「式の等しさ」は「式の表示の等しさ」と同じことです．式の等しさは $\equiv$ という記号で表します．たとえば，$3*4 \equiv 1+11$ です．なお，等しさを比べる場合，必ず型は等しいものとします．

では，等しさをきちんと定義づけていきましょう．

整数や実数などの数値型については，普通に等しさを定めます．

代数的データ型については，データコンストラクタが等しく，各構成要素が等しいとき，そのときに限り等しいとします．たとえば，Just 3 と等しいのは，Just 3 だけです．当たり前ですね．

関数 $f :: T \to U$ と $g :: T \to U$ が等しいとは，任意の $t :: T$ について $f\ t \equiv g\ t$ であるということです．関数のこのような性質を**外延性** (extensionality) といいます．関数は頻繁に扱いますから，よく慣れておく必要があります．ここで，関数を扱う関数を列挙してみましょう．

```
id  :: α → α
id x = x
const  :: α → β → α
const x y = x
(∘)  :: (β → γ) → (α → β) → (α → γ)
(g ∘ f) x = g (f x)
flip  :: (α → β → γ) → (β → α → γ)
flip f y x = f x y
($)  :: (α → β) → α → β
f $ x = f x
curry  :: ((α, β) → γ) → α → β → γ
curry f x y  = f (x, y)
uncurry  :: (α → β → γ) → (α, β) → γ
uncurry f (x, y) = f x y
infixr 9 n
infixr 0 $
```

これらについて，次のような性質が成り立っています．

$$
\begin{aligned}
&f \circ (g \circ h) \equiv (f \circ g) \circ h \\
&\mathsf{id} \circ f \equiv f \circ \mathsf{id} \equiv f \\
&f \circ \mathsf{const}\ x \equiv \mathsf{const}\ (f\ x) \\
&\mathsf{flip} \circ \mathsf{flip} \equiv \mathsf{id} \\
&\mathsf{curry} \circ \mathsf{uncurry} \equiv \mathsf{id}
\end{aligned}
$$

これらは簡単に証明できますから，きちんと確かめておいてください．この種の議論は頻繁に出てきます．

2 ボトムと再帰

さて，ここからが重要です．これから，**ボトム** (bottom) $\bot$ という値を考えます．この値は，計算が失敗したり，計算が終わっていなかったりする状況を表すと考えてください．たとえば，$\mathsf{undefined} \equiv \mathsf{error}\ \texttt{"wow"} \equiv \bot$ です．パターンマッチが失敗した場合にも $\bot$ を対応付けます．

そして重要なことに，ボトムは再帰を表すのに使えます．

一般に，再帰関数は**不動点コンビネータ** (fixed-point combinator, fixpoint combinator) fix と非再帰関数を使って表現できます．fix は

```
fix  :: (α → α) → α
fix f = f (fix f)
```

と定義できます．例えば，階乗を表す関数は

```
factorial  :: Int → Int
```

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

と書けますが，これを fix を使って，

```
factorial'      :: (Int -> Int) -> (Int -> Int)
factorial' _ 0 = 1
factorial' f n = n * f (n - 1)
factorial       :: Int -> Int
factorial       = fix factorial'
```

と表せます．よって，fix のみ考えればいいということが分かります．

では，fix はどのように考えればいいのでしょうか．実は fix f は一般に $\lim_{n \to \infty} f^n \perp$ とみなせるのです．ただし，f^n は関数合成の繰り返しを表します．たとえば $f^3 \perp$ は $f\,(f\,(f\,\perp))$ と同じことです．なお，lim という記号を使って極限を表しましたが，本当は極限の意味を考えなければならないところです．この「極限」は，第三章において「上限」として改めてきちんと扱います．Haskell の式は比較的いい性質を持っていますから，無限を適当に扱っても大抵問題ありません．以降，$\lim_{n \to \infty} f^n \perp$ を単純に $f^\infty \perp$ と書くことにします．

さて，これで再帰を扱えるようになりました．ここで，手馴らしに

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

という定義について考えてみましょう．これを fix を使って書き直せば，

```
fibs'  :: [Int] -> [Int]
fibs' s = 0 : 1 : zipWith (+) s (tail s)
fibs   :: [Int]
fibs   = fix fibs'
```

となります．そして，

$$
\begin{aligned}
&\mathsf{fibs}'\,\perp \;\equiv 0 : 1 : \perp \\
&\mathsf{fibs}'^2\,\perp \equiv 0 : 1 : \mathsf{zipWith}\,(+)\,(0 : 1 : \perp)\,(1 : \perp) \equiv 0 : 1 : 1 : \perp \\
&\mathsf{fibs}'^3\,\perp \equiv 0 : 1 : \mathsf{zipWith}\,(+)\,(0 : 1 : 1 : \perp)\,(1 : 1 : \perp) \equiv 0 : 1 : 1 : 2 : \perp \\
&\mathsf{fibs}'^4\,\perp \equiv 0 : 1 : \mathsf{zipWith}\,(+)\,(0 : 1 : 1 : 2 : \perp)\,(1 : 1 : 2 : \perp) \equiv 0 : 1 : 1 : 2 : 3 : \perp \\
&\cdots
\end{aligned}
$$

となりますから，無限にこれを続ければ，フィボナッチ数列の無限リストが出来るということが分かりますね．

3 モノイド

Haskell には「公理」を持つ型クラスがいくつかあります．そのうち最も基本的であちこちで見かけるものが**モノイド**でしょう．

```
class Monoid μ where
  ∅  :: μ
  (◇) :: μ → μ → μ
```

モノイド則は以下のとおりです．

結合性	$x \diamond (y \diamond z) \equiv (x \diamond y) \diamond z$	(Mono-Assoc)
左恒等性	$\varnothing \diamond x \equiv x$	(Mono-LIdent)
右恒等性	$x \diamond \varnothing \equiv x$	(Mono-RIdent)

簡単ですね．

型クラスに慣れるために，いろいろなモノイドを考えてみましょう．

```
instance Monoid [α] where
  ∅  = []
  (◇) = (++)

newtype First α = First (Maybe α)
instance Monoid (First α) where
  ∅                     = First Nothing
  First Nothing ◇ x     = x
  y ◇ _                 = y

newtype Last α = Last (Maybe α)
instance Monoid (Last α) where
  ∅                     = Last Nothing
  x ◇ Last Nothing      = x
  _ ◇ y                 = y

newtype Sum α = Sum α
instance Num α ⇒ Monoid (Sum α) where
  ∅                     = Sum 0
  Sum x ◇ Sum y         = Sum $ x + y

newtype Product α = Product α
instance Num α ⇒ Monoid (Product α) where
  ∅                     = Product 1
  Product x ◇ Product y = Product $ x × y

newtype Endo α = Endo (α → α)
instance Monoid (Endo α) where
  ∅                     = Endo id
  Endo f ◇ Endo g       = Endo $ f ∘ g
```

さらに，モノイドからモノイドをつくってみましょう．

```
newtype Dual α = Dual α
instance Monoid α ⇒ Monoid (Dual α) where
```

```
  ∅                 = Dual ∅
  Dual x ◇ Dual y = Dual $ y ◇ x
instance (Monoid α, Monoid β) ⇒ Monoid (α, β) where
  ∅                 = (∅, ∅)
  (a, b) ◇ (c, d) = (a ◇ c, b ◇ d)
instance Monoid β ⇒ Monoid (α → β) where
  ∅ _       = ∅
  (f ◇ g) x = f x ◇ g x
```

それぞれモノイド則を満たしていることを確認してみてください．
モノイドの感覚に慣れれば，他の型クラスに触れてもすんなり受け容れられるはずです．

第三章

データ型の深淵

1　データ型の基本

Haskell にはたくさんのデータ型があります．Int, Integer, Double などの数値型，関数型 $\alpha \to \beta$，タプル型 $(\alpha_1, \alpha_2, \cdots, \alpha_n)$，リスト型 $[\alpha]$，Bool や Maybe α や Either $\alpha\,\beta$ などのおなじみの代数的データ型，IO α や ST α などの得体のしれない型……言い出したらキリがありません．

Haskell は実用言語ですから、言語設計には利便性の問題が関わっています．とはいえ，そういう問題をいったん取っ払って，関数型と代数的データ型という一番基本的な所だけを考えても，十分多様な型を扱えます．タプル型もリスト型も代数的データ型で表現できます．インタープリタの世界で考えればとりあえず IO α や ST α のような厄介な型は必要ありません．数値型も，とりあえず無くてもなんとかなるでしょう（自然数型は代数的データ型で表現できます）．ここでは，Haskell の中核の「外」から与えられる型よりも，「内」に潜んでいる型の体系を深く深く探っていきましょう．

まずそもそも，似たようなデータ型はいろいろあります．「2 つのデータ型がだいたい同じ」というのは感覚的に扱えることも多いですが，話が複雑になってくると感覚だけではしんどいですから，データ型どうしが似ているということを正確に表現する方法が欲しいものです．そこで，**同型**という概念を利用しましょう．

型 T と U について，関数

$$f :: T \to U,\ g :: U \to T$$

が存在し，

$$g \circ f \equiv \mathsf{id},\ f \circ g \equiv \mathsf{id}$$

が満たされているとき，T と U は**同型** (isomorphic) であるといい，$T \cong U$ と書きます．

簡単にいえば，2 つの型が同型であるということは，情報を損なわずに型を相互変換できるということです．これから，同型な型どうしは積極的に同一視することにしましょう．

議論を円滑にするために，便利な用語や記号を導入しましょう．

直積型 (direct product type) $T \times U$ は Haskell でいう (T, U) にあたります．たとえば Bool $\times$ Bool $=$ (Bool, Bool) の値は

(True, True), (True, False), (False, True), (False, False)

だけです．一般に，T が n 種類の値を，U が m 種類の値を持つとき，$T \times U$ は $n \times m$ 種類の値を持ちます．まさに積ですね．

直和型 (direct sum type) $T + U$ は Haskell でいう Either T U にあたります．たとえば Bool + Bool = Either Bool Bool の値は

Left True, Left False, Right True, Right False

だけです．一般に，T が n 種類の値を，U が m 種類の値を持つとき，$T + U$ は $n + m$ 種類の値を持ちます．まさに和ですね．

関数型 (function type) U^T は Haskell でいう $T \to U$ にあたります．すこし格好をつけて冪の形で表記してみることにします．たとえば $\mathsf{Bool}^{\mathsf{Bool}}$ = Bool → Bool の値は

id, not, const True, const False

だけです（表などを書いて考えてみるといいかもしれません）．一般に，T が n 種類の値を，U が m 種類の値を持つとき，n 個それぞれについて対応させる先が m 通りあるので，U^T は m^n 種類の値を持ちます．まさに冪ですね．

さて，一般に n 個の型 $T_1, T_2, \cdots, T_n$ について，それらの直積型 $T_1 \times T_2 \times \cdots \times T_n$ を考えることができます．でも $n = 0$ の場合は何に当たるでしょうか．それがユニット型です．**ユニット型** (unit type) Unit は Haskell でいう () にあたります．**トップ型** (top type) ともいいます．Unit の値は一つしかありません．この記事ではその値を unit と書くことにします（Haskell では () にあたります）．**値が一つしかない**ということがユニット型の特徴です．

同様に，一般に n 個の型 $T_1, T_2, \cdots, T_n$ について，それらの直和型 $T_1 + T_2 + \cdots + T_n$ を考えることができます．でも $n = 0$ の場合は何に当たるでしょうか．それがボトム型です．**ボトム型** (bottom type) Bottom は Haskell でいう**データコンストラクタがない型**にあたります．データコンストラクタがない型というのは，たとえば

```
data Empty
```

のようなものです（これで定義が完結しています！ただしこのような型をつくるには EmptyDataDecls という GHC 拡張が必要です）．さて，Bottom の値は**ありません**．そんな型があってもいいのです．数学でいう空集合のようなものです．Haskell の型レベルプログラミングでも（改めてボトム型と呼ばれることは少ないですが）ボトム型は活躍しています．

以上，直積型，直和型，関数型，ユニット型，ボトム型が出揃いました．

ところで，有限個の値しか持たないデータ型については，直積型が個数の掛け算と，直和型が個数の足し算と，関数型が個数の累乗と対応していました．でも，無限個の値を持つデータ型も私たちはたくさん扱います（メモリの制限などの話はとりあえず忘れましょう）．有限と無限の違いは，関数型において特にはっきり表れます．

Int には有限個の値しかありません．4 バイト（32 ビット）であると仮定すると，2^{32} 個の値があります．Int から Int への関数型 Int → Int の値は有限個（$(2^{32})^{2^{32}}$ 個）しかありません．2^{32} 個のパターンマッチを行えば，好きな値へと対応付けられるので，（現実的であるかはともかくとして）どんな関数でもつくれます．たとえば

```
f :: Int → Int
```

$$
\begin{aligned}
&f\ 0 = 314\\
&f\ 1 = 1\\
&f\ 2 = 59265\\
&\cdots
\end{aligned}
$$

という具合です．

さて，Integer には無限個の値があります．Integer → Integer 型の関数の定義の際に無限個のパターンマッチを行うわけにはいきませんから，うまく再帰などをしないといけないというわけです．でも頑張れば好きな関数を作れるのでしょうか．ここで少し考えてみましょう．整数から整数への関数全体からなる集合は，ふつうの集合論では連続体濃度です．要は整数全体の集合の濃度（可算無限濃度）よりも大きいということです．一方，Haskell のソースコードはバイナリエディタで開いてみればただのビット列ですから，整数であるとみなせます．よって，Haskell のソースコードは可算無限個しかありませんから，整数から整数への関数を全て表現し尽くすことなんて元々出来るわけがないのです．Integer → Integer の値としては，Haskell 上で**記述可能**なもの，あるいは**計算可能**なものだけが認められるのです．

こうしてみると，関数型を集合論的な「関数全体の集合」としてみなすのはかなり問題がありそうです．いったん Haskell に寄り添って型が何を表しているか考えてみなければならないようです．Haskell を数学的に扱う際はこうした部分にトリックがあるわけです．

直積型，直和型，関数型についての基本的な関数を整理しておきましょう．

```
— 直積についての関数
(⊗)              :: (α → γ) → (β → δ) → (α, β) → (γ, δ)
(f ⊗ g) (x, y) = (f x, g y)

double   :: α → (α, α)
double x = (x, x)

swap       :: (α, β) → (β, α)
swap (x, y) = (y, x)

assocprod              :: ((α, β), γ) → (α, (β, γ))
assocprod ((x, y), z)  = (x, (y, z))
unassocprod            :: (α, (β, γ)) → ((α, β), γ)
unassocprod (x, (y, z)) = ((x, y), z)

— 直和についての関数
(⊕)              :: (α → γ) → (β → δ) → Either α β → Either γ δ
(f ⊕ _) (Left x)  = Left (f x)
(_ ⊕ g) (Right y) = Right (g y)

anyway           :: Either α α → α
anyway (Left x)  = x
anyway (Right x) = x
```

```
mirror          :: Either α β → Either β α
mirror (Left x)  = Right x
mirror (Right y) = Left y

assocsum                  :: Either (Either α β) γ → Either α (Either β γ)
assocsum (Left (Left x))   = Left x
assocsum (Left (Right y))  = Right (Left y)
assocsum (Right z)         = Right (Right z)
unassocsum                :: Either α (Either β γ) → Either (Either α β) γ
unassocsum Left x          = (Left (Left x))
unassocsum Right (Left y)  = (Left (Right y))
unassocsum Right (Right z) = (Right z)

— 直積と直和についての関数
distr                 :: (Either α β, γ) → Either (α, γ) (β, γ)
distr (Left a, c)      = Left (a, c)
distr (Right b, c)     = Right (b, c)
undistr               :: Either (α, γ) (β, γ) → (Either α β, γ)
undistr (Left (a, c))  = (Left a, c)
undistr (Right (b, c)) = (Right b, c)

— 直積と直和と関数についての関数
curry          :: ((α, β) → γ) → α → β → γ
curry f x y     = f (x, y)
uncurry        :: (α → β → γ) → (α, β) → γ
uncurry f (x, y) = f x y

fanout       :: (α → β, α → γ) → α → (β, γ)
fanout (f, g) = λx → (f x, g x)
unfanout     :: (α → (β, γ)) → (α → β, α → γ)
unfanout f    = (fst ∘ f, snd ∘ f)

fanin                 :: (α → γ, β → γ) → Either α β → γ
fanin (f, _) (Left x)  = f x
fanin (_, g) (Right y) = g y
unfanin               :: (Either α β → γ) → (α → γ, β → γ)
unfanin f              = (λx → f (Left x), λy → f (Right y))
```

見たらすぐに分かると思います．結構いろいろありますね．これらの関数を眺めていると，同型な型がいろいろ見つかります．

$$
\begin{aligned}
(T \times U) \times V &\cong T \times (U \times V) && \because \text{assocprod, unassocprod} \\
T \times U &\cong U \times T && \because \text{swap} \\
(T + U) + V &\cong T + (U + V) && \because \text{assocsum, unassocsum} \\
T + U &\cong U + T && \because \text{mirror} \\
(T + U) \times V &\cong T \times V + U \times V && \because \text{distr, undistr}
\end{aligned}
$$

$$
\begin{aligned}
V^{T \times U} &\cong (V^U)^T && \because \text{curry, uncurry} \\
U^T \times V^T &\cong (U \times V)^T && \because \text{fanout, unfanout} \\
V^T \times V^U &\cong V^{T+U} && \because \text{fanin, unfanin}
\end{aligned}
$$

こうしてみると，整数や実数について成り立っているような結合則，分配則，指数法則が成り立っています．これは偶然ではなくて，直積型や直和型や関数型は綺麗な性質を持っているのです．

再帰的でない代数的データ型は直積型と直和型だけで表現することができます．

$$
\begin{aligned}
&\textbf{data}\ D\ \alpha_1 \cdots \alpha_k \\
&\quad = C_1\ T_{1,1}\ T_{1,2} \cdots T_{1,n_1} \\
&\quad \mid C_2\ T_{2,1}\ T_{2,2} \cdots T_{2,n_2} \\
&\quad \cdots \\
&\quad \mid C_m\ T_{m,1}\ T_{m,2} \cdots T_{m,n_m}
\end{aligned}
$$

という代数的データ型があるとき，

$$
\begin{aligned}
&D\ \alpha_1 \cdots \alpha_k \\
&\quad \cong (T_{1,1} \times T_{1,2} \times \cdots \times T_{1,n_1}) + (T_{2,1} \times T_{2,2} \times \cdots \times T_{2,n_2}) \\
&\qquad\qquad + \cdots + (T_{m,1} \times T_{m,2} \times \cdots \times T_{m,n_m})
\end{aligned}
$$

となります．端的にいえば，代数的データ型は「直積の直和」なのです．

次は再帰的データ型について考えてみましょう．

一番基本的な再帰的データ型といえばやはりリスト型ですね．Haskell ではリスト型はプリミティヴに扱われていますが，もし実際に定義できるとしたら

$$
\textbf{data}\ [\alpha] = [] \mid \alpha : [\alpha]
$$

のようになるでしょう．普通に定義するならば

$$
\textbf{data}\ \mathsf{List}\ \alpha = \mathsf{Nil} \mid \mathsf{Cons}\ \alpha\ (\mathsf{List}\ \alpha)
$$

みたいになるでしょう．

自然数型も実は再帰的データ型として定義できます．

$$
\textbf{data}\ \mathsf{Nat} = \mathsf{Zero} \mid \mathsf{Succ}\ \mathsf{Nat}
$$

例えば，Succ (Succ (Succ Zero)) は 3 を表します．自然数型 Nat とユニットのリスト [()] が同型であることにも注目してください．

二分木型や多分木などの木構造も再帰的データ型として定義できます．[1]

$$
\textbf{data}\ \mathsf{Tree}\ \alpha = \mathsf{Leaf} \mid \mathsf{Node}\ (\mathsf{Tree}\ \alpha)\ \alpha\ (\mathsf{Tree}\ \alpha)
$$

[1] ちなみに，多分木を英語で rose tree というのは，[1] によると，ミアテンス氏が “rhododendron”（シャクナゲ）というギリシア語由来の言葉を英語風に直したものらしいです．バラの木よりもシャクナゲの木のほうがボリュームがあって多分木らしい気もします．

$$\textbf{data}\ \mathsf{Rose}\,\alpha = \mathsf{Branch}\,\alpha\,[\mathsf{Rose}\,\alpha]$$

このようにいろいろな再帰的データ型がありますが，再帰的データ型を作るための最小限の機能はなんでしょうか．それは**不動点コンビネータ**です．再帰的な関数は不動点コンビネータ fix だけで表現できましたが，型についても同様に不動点コンビネータが考えられるのです．

$$\textbf{newtype}\ \mathsf{Fix}\,\varphi = \mathsf{In}\,\{\mathsf{out} :: \varphi\,(\mathsf{Fix}\,\varphi)\}$$

ここで，$\varphi\,(\mathsf{Fix}\,\varphi) \cong \mathsf{Fix}\,\varphi$ という性質が非常に重要です．このコンビネータを使えば，さまざまな再帰的データ型を非再帰的データ型によって表現できます．たとえば，

$$\textbf{data}\ \mathsf{UList}\,\alpha\,\tau = \mathsf{UNil} \mid \mathsf{UCons}\,\alpha\,\tau$$
$$\textbf{type}\ \mathsf{List}\,\alpha = \mathsf{Fix}\,(\mathsf{UList}\,\alpha)$$

というふうにリスト型を定義すればいいのです．$\mathsf{UList}\ T$ のようなデータ型を $X \mapsto \mathrm{Unit} + T \times X$ と書き，不動点コンビネータを Fix と書くことにすると，T のリスト型は $\mathrm{Fix}(X \mapsto \mathrm{Unit} + T \times X)$ と書けます．

これまでだいぶ主要な要素は出揃いましたが，モダンな Haskell では**全称量化型** (universally quantified type) も重要です．従来の Haskell ではごく限定された形でしか全称量化型を扱えませんでしたが，RankNTypes という GHC 拡張が追加されて，全称量化型を自由に使えるようになりました．

まず，恒等関数 id の型の全称量化を明示的に書けば，

$$\mathsf{id} :: \forall\,\alpha.\,\alpha \to \alpha$$

となります．従来の Haskell では，このように定義の一番外側にだけ $\forall$ を付けられます．

一方，RankNTypes という GHC 拡張を使えば，全称量化型をいくらでも入れ子にできます．たとえば，実際に使う可能性のある関数としては

$$\mathsf{runST} :: \forall\,\alpha.\,(\forall\,\sigma.\,\mathsf{ST}\,\sigma\,\alpha) \to \alpha$$

というものがあります．この関数は全称量化型を使うことによって，巧妙にスレッドを保護しています．

全称量化型は今後の議論で多用していきます．

2 関手に親しもう

データ型を扱う上での基本中の基本が**関手** (functor) です．関手についてじっくり学びましょう．

関手の普遍性を考えるために，ここでいったん，**部分型付け** (subtyping) について考えてみます．

Haskell は暗黙の型変換が行われない，部分型付けの無い言語ですが，主要なプログラミング言語を見ると部分型付けが行われる言語は非常に多いです．型 T が型 U の**部分型** (subtype) であるとは，型 T の値を，型 U の値へと情報を損なわない形で変換できるということです．T が U の部分型であり，変換に使われる関数が $f :: T \to U$ であるということを $T \subset_f U$ と書くことにします．C 言語などの多くの言語では，本来は実数型を受け取る sin のような関数に 3 などの整数値を渡しても，勝手に実数値に変換されますし，部分型は日常的に使っていることだと思います（むしろ Haskell のような，部分型付

けを全く行わない言語のほうが珍しいかもしれません）．Haskell 上に部分型付けのシステムを構築してみるつもりで考えてみましょう．

たとえば，Int は Double の部分型にしたいです．変換関数は fromIntegral :: Int → Double です．つまり，Int $\subset_{\mathsf{fromIntegral}}$ Double です．さらに便利に使うためにいろいろと拡張したいですね．たとえば，(Int, Int) は (Double, Double) の部分型にしたいですし，Either Int Int は Either Double Double の部分型にしたいですし，[Int] は [Double] の部分型にしたいです．

関数型についてはどうでしょうか？よく考えると，Double → Int が Int → Double の部分型になってくれたらうれしいです．関数の引数側についてはなんだか逆向きになっていて，引数側がより小さく，返り値側がより大きいほうが，関数型としても大きくなるのです．ちょっと例を挙げてみましょう．exponent :: Double → Int という関数は浮動小数点数を受け取ってその桁数を返す関数です．Double 型の実数値をこの関数に渡せるということは，Int 型の整数値を渡せるということですから，exponent′ :: Int → Int という関数を作れます．さらに，返り値の整数値を（すこし奇妙ですが）実数値として返すこともできるので，exponent″ :: Int → Double という関数を作れます．Double → Int という型の関数から Int → Double という型の関数を得られるのです．

一般的には，次のようにすればよさそうです．

$A \subset_f A'$ かつ $B \subset_g B'$ のとき，$(A, B) \subset_{f \otimes g} (A', B')$
$A \subset_f A'$ かつ $B \subset_g B'$ のとき，Either $A\ B \subset_{f \oplus g}$ Either $A'\ B'$
$A \subset_f A'$ かつ $B \subset_g B'$ のとき，$A' \to B \subset_{\lambda k \to g \circ k \circ f} A \to B'$
$A \subset_f A'$ のとき，$[A] \subset_{\mathsf{map}\, f} [A']$

もっとほかのデータ型についてもいろいろ部分型付けの規則が欲しいですね．

実はここにすでに「関手のタネ」が潜んでいるのです．

一般に，データ型の間で，基本的構造は保ったまま，値だけを変換するのが関手です．部分型付けでは，ふつう情報を損なわない変換しか考えませんが，そんな制約を取り払ってしまえば関手になるのです．

関手にはいろいろあります．

いちばん有名なのが普通の関手です．特に他の関手と区別するときは**共変関手** (covariant functor) と呼ぶこともあります．これは見慣れていると思います．

```
class Functor (φ :: * → *) where
  fmap :: (α → β) → (φ α → φ β)
(⟨$⟩) :: Functor φ ⇒ (α → β) → (φ α → φ β)
(⟨$⟩) = fmap
infixl 4 ⟨$⟩
```

関手則は次の通りです．

恒等 fmap id ≡ id　　(Fmap-Id)
合成 fmap $(g \circ f)$ ≡ fmap $g \circ$ fmap f　　(Fmap-Comp)

見た目にも綺麗な法則ですからすぐに受け入れられると思います．関手になるデータ型はいろいろあります．

```
instance Functor [] where
  fmap = map
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
instance Functor Tree where
  fmap _ Leaf         = Leaf
  fmap f (Node l x r) = Node (fmap f l) (f x) (fmap f r)
instance Functor Rose where
  fmap f (Branch x t) = Branch (f x) (map (fmap f) t)
```

型に注目してみてください．また，再帰的なデータ型は fmap の定義も再帰的になっているということも少し注目してください．

引数を二つ取るデータ型については，**双関手** (bifunctor) があります．

```
class Bifunctor (φ :: * → * → *) where
  bimap :: (α → α') → (β → β') → (φ α β → φ α' β')
```

双関手則は次の通りです．

恒等 bimap id id ≡ id (Bimap-Id)

合成 bimap (g ∘ f) (k ∘ h) ≡ bimap g k ∘ bimap f h (Bimap-Comp)

これも関手則と似ていますからすぐ分かると思います．双関手であるデータ型も結構あります．

```
instance Bifunctor (,) where
  fmap = (⊗)
instance Bifunctor Either where
  fmap = (⊕)
```

なお，双関手の第一引数を固定するか，第二引数を固定すると，関手になります．

関数型については，引数側がなんだか逆になっていました．これは**プロ関手** (profunctor) のひとつです．[2]

```
class Profunctor (φ :: * → * → *) where
  dimap :: (α' → α) → (β → β') → (φ α β → φ α' β')
```

プロ関手則は次の通りです．

恒等 dimap id id ≡ id (Dimap-Id)

合成 dimap (g ∘ f) (k ∘ h) ≡ dimap f k ∘ dimap g h (Dimap-Comp)

「合成」の方は少し入れ替わっていますが，型を考えれば納得できると思います．さて，関数をプロ関手にしておきましょう．

[2] しっくり来る日本語訳がないのでとりあえず「プロ関手」としておきます．

```
instance Profunctor (→) where
  dimap f g k = g ∘ k ∘ f
```

プロ関手の第二引数を固定すると，**反変関手** (contravariant functor) になります．

```
class Contravariant (φ :: * → *) where
  contramap :: (α' → α) → (φ α → φ α')
```

反変関手則は次の通りです．

恒等 contramap id ≡ id (Contra-Id)

合成 contramap (g ∘ f) ≡ contramap f ∘ contramap g (Contra-Comp)

これも型を考えれば納得できると思います．身近な反変関手はないので，関数型の返り値側の引数を固定して反変関手を作ることにします．

```
newtype Op α β = Op (β → α)
instance Contravariant (Op α) where
  contramap f (Op k) = Op (k ∘ f)
```

ここまでを整理すると，一引数のデータ型には**共変関手**になるものもあるし，**反変関手**になるものもあります．二引数のデータ型について，**双関手**は第一引数についても第二引数についても**共変**であり，**プロ関手**は第一引数については**反変**であり，第二引数については**共変**です．特に名前はありませんが，**反変**と**共変**の組み合わせ，**反変**と**反変**の組み合わせもあります．これまでを踏まえれば，この二つのそれぞれについても，型クラスとその満たすべき規則を作ることができますね．同様に考えれば，三引数のデータ型については 8 種類の型クラス，四引数のデータ型については 16 種類の型クラス，……とどんどん考えていける訳ですが，どんなに引数が多くても，ある一つの引数以外を（先ほどの Op のように）固定すれば，一引数のデータ型に帰着されますから，共変関手と反変関手のみについて考えていきましょう．

この違いは**正の位置**と**負の位置**という概念で説明できます．正の位置を + で，負の位置を − で表すことにすると，直積型，直和型，関数型について

```
+ × +
+ + +
− → +
```

となっています．これを組み合わせると，符号のかけ算が起こります．

```
(+ × +) + (+ × +)
(− + −) → (+ + +)
(− → +) × (− → +)
(+ → −) → +
```

どういう仕組みかはもうわかったと思います．

データ型の定義において，注目している引数が正の位置のみに現れているなら共変関手になり，負の位置のみに現れているなら反変関手になるのです．正の位置と負の位置については後で別の形で説明し直します．

関手の一般的な作り方をもう少し考えてみましょう．
再帰的データ型については，次のようにして関手にできます．

```
instance Functor φ ⇒ Functor (Fix φ) where
  fmap f (In x) = In (fmap (fmap f) x)
```

結構簡単ですね．
データ型の合成について考えてみましょう．関数合成と同様，データ型の合成もすんなり定義できます．

```
newtype (φ :∘ φ') α = Comp {unComp :: φ (φ' α)}
```

ここで，$(\varphi :\circ \varphi')\ \alpha$ というのは $(:\circ)\ \varphi\ \varphi'\ \alpha$ と同じことです．
さて，共変関手と共変関手を合成すると共変関手に，共変関手と反変関手を合成すると反変関手に，反変関手と共変関手を合成すると反変関手に，反変関手と反変関手を合成すると共変関手になります．とりあえず反変関手と反変関手の合成だけ挙げておきましょう．

```
instance (Contravariant φ, Contravariant φ') ⇒ Functor (φ :∘ φ') where
  fmap f (Comp x) = Comp (contramap (contramap f) x)
```

関手になることを確かめてみてください．
継続モナドもこのメソッドを使えばすぐに関手に出来ます．

```
newtype Cont ρ α = Cont ((α → ρ) → ρ)
instance Functor (Cont ρ) where
  fmap f (Cont c) = Cont (c ∘ (∘ f))
```

ここは納得するまできちんと考えてみてくださいね．

3 System F をつくろう

Haskell で議論をすることもできますが，Haskell はそのままだと扱いにくいので，*System F*（システム・エフ）を構築し，その上で議論をしていきます．Haskell にはたくさんの便利な機能がありますが，その多くは議論の本質には無関係です．特に Haskell の型推論の機構は比較的複雑で，これも議論の妨げとなっています．そこで，System F というミニマムな計算体系を構築していくのです．Haskell も System F もベースはラムダ計算であり，本質部分は非常に似ていますから，Haskell（の部分言語）と System F との対応関係は比較的簡単に分かるはずです．しばらく苦しいかもしれませんが，辛抱強く読んでいってください．ここからの目標は，**自然性**という重要な性質を導いて行くことです．少々長い道のりですが，定義さえきちんと行えば証明は素直なので，大丈夫です．

System F は，**多相ラムダ計算** (polymorphic lambda calculus) や**ジラール・レナルズ型システム** (Girard-Reynolds type system) や**二階ラムダ計算** (second-order lambda calculus) とも呼ばれています．いろいろ名前が付けられることからも分かるように，かなり基本的で重要な計算体系です．

System F の位置づけを概観しておきましょう．ラムダ計算のうち最も基本的なものが**型無しラムダ計算** (untyped lambda calculus) です．そこに型付けを加えたもののうち最も基本的なものが（その名も）**単純型付きラムダ計算** (simply typed lambda calculus) です．このラムダ計算は，**関数型** (function type) といくつかの型定数のみから型が構成されています．さて，そこにさらに**全称量化型** (universally quantified type) を加えたのが System F なのです．

これから「つくる」System F の特徴をまとめておきます．

1. 関数型・全称量化型・型変数のみから型が構成されている．
   議論を簡単にするために，型システムは**ミニマム**にしてあります．もちろん直和型，直積型，自然数型，リスト型などの様々なデータ型をさらに加えても構いませんが，ここでつくる System F には関数型と全称量化型しかありません．
   これだけでは不安かもしれませんが，この型システムは驚くべきポテンシャルを秘めています．あとできちんと直積型，直和型，再帰型などのデータ型をつくるので，安心してください．
2. 型注釈がいちいち必要である．大文字のラムダや添字も使う．
   Haskell や ML のような強力な型推論器付きの体系だと，型推論器の記述に手間がかかります．また，型を省略するのは，ここでの議論ではほぼ無益で，時に有害ですらあります．そこで，型注釈をきちんとつけることにします．さらに，型を量化していることを明示するために大文字のラムダを使ったり，量化された型を指定するために添字を使ったりします．

Haskell と少し違いますが，Haskell に慣れているならばここでの記法にもすぐに慣れるはずです．

なお，Haskell の型システムのベースは**ヒンドリー・ミルナー型システム** (Hindley-Milner type system) と呼ばれています．この型システムでは *Algorithm W*（アルゴリズム・ダブリュー）と呼ばれるアルゴリズムで高速に型推論をすることができます．さらに，**任意ランク多相**という GHC 拡張（いわゆる RankNTypes）を使っている場合，System F と同様の機能を持つようになります．このとき，型推論は本来決定不能ですが，「ラムダ束縛された変数や case 束縛された変数は，多相的な型を明示的に与えない限り，その型が forall を含まない」とみなすことによって，型推論アルゴリズムを決定可能にしています．

それでは System F をつくっていきましょう．なお，以下の議論はワドラー氏の論文 [12] に基づいています．

System F の構文にあるのは，大まかに言えば，**型変数** (type variable)，**型** (type)，**変数** (variable)，**項** (term) だけです．[3]

$$
\begin{aligned}
(\text{型変数}) &::= X \mid Y \mid Z \mid \ldots \\
(\text{型}) &::= (\text{型変数}) \mid (\text{型}) \to (\text{型}) \mid \forall(\text{型変数}).\ (\text{型}) \\
(\text{変数}) &::= x \mid y \mid z \mid \ldots \\
(\text{項}) &::= (\text{変数}) \mid \lambda\,(\text{変数}) : (\text{型}).\ (\text{項}) \mid (\text{項})\ (\text{項}) \mid \Lambda(\text{型変数}).\ (\text{項}) \mid (\text{項})_{(\text{型})}
\end{aligned}
$$

Haskell にない表記もありますが，おいおい説明します．大文字のラムダ「Λ(型変数). (項)」と型の添字「$(\text{項})_{(\text{型})}$」は見慣れないかもしれません．これは，それぞれ「型の量化」と「量化された型の指定」を項において明示的に行うための記法です．[4] 型推論の手間を省くためにこの記法を導入しています．例えば，恒等関数 id は $\Lambda X.\, \lambda\, x : X.\, x : \forall X.\, X$ となり，それを型 T の項 t に適用したい場合は $\mathrm{id}_T\ t$ と

[3] 型変数と変数は別個のものです．

[4] Haskell だとこのようなことは型推論で暗黙的に行われます．逆に型注釈を使わず明示する方法がありません．

なります．Haskell では $\forall X.\, \forall Y.\, X \to Y$ も $\forall Y.\, \forall X.\, X \to Y$ も全く同じですが，ここでつくる System F では両者を区別するので注意してください．なお，略記として，$(t_T)_U$ を $t_{T,U}$ と書くことにします．

型変数を X, Y, Z で，型を T, U, V で，変数を x, y, z で，項を t, u, v で表すことにします．ダッシュ（プライム）記号を付けている場合もあります．

自由型変数 (free type variable) と**自由変数** (free variable) という言葉も導入しておきましょう．「自由 (free)」というのは内部で λ や Λ や $\forall$ によって「束縛 (bind)」されていないということです．型 T の自由型変数全体の集合を $\mathrm{Fv}(T)$，項 t の自由型変数全体の集合を $\mathrm{Fv}(t)$，自由変数全体の集合を $\mathrm{fv}(t)$ と表記します．例えば，

$$\begin{aligned}\mathrm{Fv}(X \to (\forall Y.\, Z \to Y)) &= \{X, Z\}\\ \mathrm{Fv}(\Lambda X.\, \lambda\, x : X \to Y.\, z_Z(x\ y)) &= \{Y, Z\}\\ \mathrm{fv}(\Lambda X.\, \lambda\, x : X \to Y.\, z_Z(x\ y)) &= \{y, z\}\end{aligned}$$

となります．自由型変数を持たない型を**閉じた型** (closed type) といい，自由型変数および自由変数を持たない項を**閉じた項** (closed term) ということにします．

置換 (substitution) も定義しておきましょう． $T[U/X]$ は型 T の自由型変数 X をすべて型 U に置き換えてできる型です．$t[u/x]$ は項 t の自由変数 x をすべて項 u に置き換えてできる項です．$t[U/X]$ は項 t の自由型変数 X をすべて型 U に置き換えてできる項です．

準備は整いました．**型推論** (type inference) のルールを定義していきます．なお，型推論のルールを使って**型検査** (type checking) ができます．型推論と型検査は共通する要素がかなり多いです．

項 t が型 T を持つ，ということが導出されるということを，次のように書きます．[5]

$$\vec{X}; \vec{x} \vdash t : T$$

ここで，$\vec{X}$ は型変数の集合 $\{X_1, X_2, \ldots, X_n\}$ であり，利用できる自由型変数（あるいはこれまでに Λ により束縛された型変数）を表します．$\vec{x}$ は変数と型の組の集合 $\{\langle x_1, T_1\rangle, \langle x_2, T_2\rangle, \cdots, \langle x_m, T_m\rangle\}$ であり，利用できる自由変数（あるいはこれまでに λ により束縛された変数）と対応する型を表します（単なる変数の集合ではないので注意してください）．なお，$\vec{X}$ や $\vec{x}$ で何も要素を持たないものは（空集合なので）$\varnothing$ と書くことにします．種類にかかわらず，何も要素を持たないならば $\varnothing$ と書きます．

なお，$\vec{X}; \vec{x} \vdash t : T$ と書く場合，常に

$$\text{「}\mathrm{Fv}(T) \subset \vec{X}\text{」かつ「}\mathrm{Fv}(t) \subset \vec{X}\text{」かつ「}\mathrm{fv}(t) \subset \{x_j | \langle x_j, T_j\rangle \in \vec{x}\}\text{」}$$

が成り立っているものとします．

では，型推論のルールを列挙していきましょう．長い横棒は，「上の条件が（すべて）満たされているならば，かつそのときに限り，下の条件が満たされる」ということを表しています．[6]

$$\frac{\langle y, T\rangle \in \vec{x}}{\vec{X}; \vec{x} \vdash y : T} \qquad \text{(Var-Trivial)}$$

[5] この $\vdash$ という記号は，その形にちなんで，ターンスタイル (turnstile; 自動改札機) やティー (tee; アルファベットの T) やライトタック (right tack; 右向きの画鋲) などと呼ばれます．また，TEX においては \vdash (vertical bar + dash) というコマンドで出てきます．

[6] 普通は「…満たされているならば，下の条件が満たされる」ということを表しますが，このルールの集まりでは，そこから「…満たされているならば，かつそのときに限り，下の条件が満たされる」ということが導かれるので，最初からそうであると定義しておきます．

$$\frac{\vec{X};\vec{x},\langle y:U\rangle \vdash v:V \qquad \mathrm{Fv}(U)\subset\vec{X}}{\vec{X};\vec{x}\vdash(\lambda\, y:U.\, v):U\to V} \quad \text{(Func-Intro)}$$

$$\frac{\vec{X};\vec{x}\vdash t:U\to V \qquad \vec{X};\vec{x}\vdash u:U}{\vec{X};\vec{x}\vdash(t\, u):V} \quad \text{(Func-Elim)}$$

$$\frac{\vec{X},Y;\vec{x}\vdash t:T}{\vec{X};\vec{x}\vdash(\Lambda Y.\, t):\forall Y.\, T} \quad \text{(Forall-Intro)}$$

$$\frac{\vec{X};\vec{x}\vdash t:\forall Y.\, T \qquad \mathrm{Fv}(U)\subset\vec{X}}{\vec{X};\vec{x}\vdash t_U:T[U/Y]} \quad \text{(Forall-Elim)}$$

なお，「$\vec{X},Y$」は「$\vec{X}\cup Y$」という意味です．また，「$\vec{x},\langle y:U\rangle$」は「$\{\langle x_j,T_j\rangle\in\vec{x}\mid x_j\neq y\}\cup\{\langle y:U\rangle\}$」の略記です（要は変数の型情報が上書きされるということです）．

ちなみに，型推論のルールを見比べると関数と全称量化型の扱いはそっくりですね．Haskell を使っていると型が暗黙に扱われるので気付きにくいですが，全称量化型は「型から型への関数」のようなものなのです．更に「型の型」から「型の型」への関数や「型の型の型」から「型の型の型」への関数なども作れるように拡張できます．そうすると**高階ラムダ計算** (higher-order lambda calculus) が作れます．

計算できないと意味が無いので簡約・変換を定義しておきます．

$$\begin{aligned}(\lambda\, x:U.\, t)u &\Rightarrow_\beta t[u/x]\\ (\Lambda X.\, t)_U &\Rightarrow_\beta t[U/X]\\ \lambda\, x:U.\, t\, x &\Rightarrow_\eta t\\ \Lambda X.\, t_X &\Rightarrow_\eta t\end{aligned}$$

ここで，$\Rightarrow_\beta$ は**β簡約**（ベータ） (beta reduction)，$\Rightarrow_\eta$ は**η変換**（イータ） (eta conversion) を表しています．簡約・変換を施せる箇所が複数ある場合もあります．このようなとき，簡約・変換を施す順番についてさまざまな選択があります．ここではいわゆる**最外簡約** (outermost reduction) をすることとしておきます．詳細は本筋には関係無いので省略しておきます．

4 意味論をつくろう

これまで，System F の「記号」だけを問題にしてきました．一般に，計算体系について記号の組み合わせについて論じる分野や理論体系を**構文論** (syntax, syntactics, syntactic theory) といいます．いっぽう，記号が持つ「意味」について論じる分野や理論体系を**意味論** (semantics, semantic theory) といいます．「意味」というと科学らしくないかもしれませんが，単純に「具体的な数学的対象」に落とし込むということだと捉えても構わないと思います．構文論と意味論を一緒くたにして扱えれば嬉しいのですが，実際にはそう上手く対応してくれません．構文論の世界と意味論の世界をきっちり区別することが重要です．言葉だけ聞くとなんだかよく分からないかもしれませんが，見ていけば分かると思います．論理学を少しかじってみるのもいいかもしれません．

“syntax” という言葉は単に「構文」という意味でプログラマも日常的に使いますが，“semantics” という言葉は馴染みが無い人も多いかもしれません．実は，意味論をあらわす英語の “semantics” は，仏教

用語の「禅」という言葉の遠い親戚に当たるようです.[7] 印欧祖語の語根 “dheie” (「見る」の意) がギリシア語で “sema” (「印」) となり，その派生語として “semantikos” (「意味を表す」) が生まれ，フランス語の “sémantique” を経て英語の “semantics” となりました．一方，“dheie” がサンスクリット語では “dhyana” (「瞑想」) となり，中国語で「禅那」と音写されて，略して「禅」と呼ばれているのです．日本人として “semantics” という言葉に少し親近感が湧きますね．

閑話休題，System F に対する意味論を作っていきましょう．これから作る意味論は，特に「**表示的意味論**」と呼ばれる種類のものです．

先に全体像をまとめておくと，構文論上の「型」を「型環境」を媒介して意味論上の「型値」に翻訳し，構文論上の「項」を「環境」を媒介して意味論上の「値」に翻訳します.「翻訳」のことを意味論において一般的には「解釈」と言うのです．抽象的な話が続きますが，実際にどう翻訳するのか，その翻訳がどれくらい上手く行っているのかに注目してください．

まず型から考えましょう．型の**モデル** (model) は，**型値** (type value) の全体からなる集合 $\mathbf{U}$ と演算 $\to\colon \mathbf{U}\times\mathbf{U}\to\mathbf{U}$ と $\forall\colon(\mathbf{U}\to\mathbf{U})\to\mathbf{U}$ の 3 つ $\langle\mathbf{U},\to,\forall\rangle$ から成ります．[8] つまり，任意の 2 つの型値 $A\in\mathbf{U},B\in\mathbf{U}$ について $(A\to B)\in\mathbf{U}$ であり，任意の型値間の関数 $F\in(\mathbf{U}\to\mathbf{U})$ について $\forall F\in\mathbf{U}$ です．以降，このモデルにおいて議論していきます．また，型値は A,B,C,D で，型値の間の関数は F で表します．

ただし，記号を多少使い回しているので注意して下さい．微妙に違う記号や記法を使い分けるということもできなくは無いですが，目に優しいほうがいいので，$\to$ や $\forall$ といった普通の記号を使います．慣れればまず意味を取り違えることはないはずです．

まず $\to$ について．System F の構文において $T\to U$ は「関数型」を表します．$A\to B$ は A から B への関数ではなく，特定の「型値」を表します．後に出てきますが，$\mathcal{A}\to\mathcal{B}$ は特定の「二項関係」を表します．それ以外の場面において，$\to$ は集合論的な意味での「関数全体からなる集合」を表しています．$f:K\to L$ というのは $f\in K\to L$ と同じことになります．たとえば，$\mathbf{U}\to\mathbf{U}$ は $\mathbf{U}$ から $\mathbf{U}$ への関数全体からなる集合を，(後に出てきますが) $\mathbf{D}_A\to\mathbf{D}_B$ は $\mathbf{D}_A$ から $\mathbf{D}_B$ への関数全体からなる集合を表します．

次に $\forall$ について．System F の構文において $\forall X.\,T$ は「全称量化型」を表します (Haskell でも同様の記法を使いますね)．$\forall F$ は特定の「型値」を表します．後に出てきますが，$(\forall A:\mathbf{U}.\,\mathbf{D}_{F(A)})$ は「型から条件を満たす値への関数全体からなる集合」を表します (後できちんと定義します)．これも後に出てきますが，$\forall\mathcal{F}$ は特定の「二項関係」を表します．

さて，型変数の集合 $\vec{X}$ に対する**型環境** (type environment) $\vec{A}$ とは，$X_i\in\vec{X}$ を $\vec{A}\,[\![X_i]\!]\in\mathbf{U}$ に対応付けるもののことです．連想配列のように考えたら分かりやすいかもしれません．さて，型 T の自由型変数をすべて含む集合 $\vec{X}$ に対する型環境 $\vec{A}$ があるとき，T の $\vec{A}$ における**解釈** (interpretation) $[\![T]\!]\,\vec{A}\in\mathbf{U}$ が次のようにして再帰的に定義できます．

$$
\begin{aligned}
[\![X]\!]\,\vec{A} &\triangleq \vec{A}\,[\![X]\!]\\
[\![T\to U]\!]\,\vec{A} &\triangleq [\![T]\!]\,\vec{A}\to[\![U]\!]\,\vec{A}\\
[\![\forall Y.\,T]\!]\,\vec{A} &\triangleq \forall(B\mapsto[\![T]\!]\,\vec{A}[B/Y])
\end{aligned}
$$

なお，$[\![\ldots]\!]$ という括弧を使うのは，「意味論の世界において ... という構文論の世界を扱っている」と

[7] Online Etymology Dictionary http://www.etymonline.com/ を元に記述しています．

[8] 「型値」は造語です．型に割り当てられる予定の数学的対象のことです.「かたち」とでも読んでください．

いうことを明示するためです．今後もたくさん使います．

ここで，$\mapsto$ というのは関数を表します．例えば，$f(x) = x+1$ は $f = x \mapsto x+1$ とも書き表せます．ラムダ式で書いてもいいのですが，意味論の話だと明示するためにあえてこの記号を使います．また，$\vec{A}[B/X]$ は，$\vec{A}$ を拡張したもの（いわゆる「**置換**」）で，

$$\vec{A}[B/Y]\,[\![X]\!] \triangleq \begin{cases} B & (X = Y) \\ \vec{A}\,[\![X]\!] & (X \neq Y) \end{cases}$$

と定義されます．

なお，$\varnothing[B/Y]$ を $[B/Y]$ と略記し，$\mathcal{A}[B/Y][C/Z]$ を $\mathcal{A}[B/Y, C/Z]$ と略記することにします．類似の記号についても同様の略記法を適用します．

次は項を含めた**モデル**を作りましょう．型のモデルは引き続き $\langle \mathbf{U}, \to, \forall \rangle$ とします．任意の型値 $A \in \mathbf{U}$ について，その要素の**値** (value) の全体からなる集合 $\mathbf{D}_A$ があるとします．なお，任意の異なる型値 A, B について $\mathbf{D}_A$ と $\mathbf{D}_B$ は共通部分を持たないとします．[9] 値全体の集合を $\mathbf{D}^*$ とすると，$\mathbf{D}$ は $\mathbf{U}$ から $\mathbf{D}^*$ への関数であると見なすこともできます．以降，値は a, b, c, d で表します．

ただし，$\mathbf{D}_{A\to B}$ が $(\mathbf{D}_A \to \mathbf{D}_B)$ と一対一に対応しているとします．つまり，関数

$$\phi_{A,B} : \mathbf{D}_{A\to B} \to (\mathbf{D}_A \to \mathbf{D}_B)$$
$$\psi_{A,B} : (\mathbf{D}_A \to \mathbf{D}_B) \to \mathbf{D}_{A\to B}$$

が存在し，$\phi_{A,B} \circ \psi_{A,B} = \mathrm{id}_{\mathbf{D}_A \to \mathbf{D}_B}$ かつ $\psi_{A,B} \circ \phi_{A,B} = \mathrm{id}_{\mathbf{D}_{A\to B}}$ を満たしているとします．

ここで，$F \in (U \to U)$ について，$(\forall A : \mathbf{U}.\, \mathbf{D}_{F(A)})$ を，

$$(\forall A : \mathbf{U}.\, \mathbf{D}_{F(A)}) \triangleq \{f : \mathbf{U} \to \mathbf{D}^* \mid \text{任意の } A \in \mathbf{U} \text{ について } f(A) \in \mathbf{D}_F(A)\}$$

と定義します．要は型値 $A \in \mathbf{U}$ を受け取って対応する型値に属する値 $a \in \mathbf{D}_{F(A)}$ を返す関数からなる集合です．

そして，$\mathbf{D}_{\forall F}$ が $(\forall A : \mathbf{U}.\, \mathbf{D}_{F(A)})$ と一対一に対応しているとします．つまり，関数

$$\Phi_F : \mathbf{D}_{\forall F} \to (\forall A : \mathbf{U}.\, \mathbf{D}_{F(A)})$$
$$\Psi_F : (\forall A : \mathbf{U}.\, \mathbf{D}_{F(A)}) \to \mathbf{D}_{\forall F}$$

が存在し，$\Phi_F \circ \Psi_F = \mathrm{id}_{(\forall A:\mathbf{U}.\ \mathbf{D}_{F(A)})}$ かつ $\Psi_F \circ \Phi_F = \mathrm{id}_{\mathbf{D}_{\forall F}}$ を満たしているとします．

いま，項 t があり，$\vec{X}; \vec{x} \vdash t : T$ が成り立っているとします．$\vec{X}$ と $\vec{x}$ にかんする**環境対** (environment pair) $\langle \vec{A}, \vec{a} \rangle$ は，$\vec{X}$ に対する型環境 $\vec{A}$ と，**環境** (environment) $\vec{a}$，つまりそれぞれの $\langle x_j, T_j \rangle \in \vec{x}$ について x_j を $\vec{a}\,[\![x_j]\!] \in \mathbf{D}_{[\![T_j]\!]\,\vec{A}}$ に対応付けるもの，この 2 つから成ります．[10] そして，項 t の環境対 $\langle \vec{A}, \vec{a} \rangle$ における**解釈** $[\![t]\!]\,\vec{A}\,\vec{a}$ が次のようにして再帰的に定義できます．[11]

$$[\![y]\!]\,\vec{A}\,\vec{a} \triangleq \vec{a}\,[\![y]\!]$$
$$[\![\lambda\, y : U.\, v]\!]\,\vec{A}\,\vec{a} \triangleq \psi_{[\![U]\!]\,\vec{A},C}\,(b \mapsto [\![v]\!]\,\vec{A}\,\vec{a}[b/y]) \qquad (\text{ただし } C \text{ は } [\![v]\!]\,\vec{A}\,\vec{a}[b/y] \in \mathbf{D}_C \text{ を満たす})$$
$$[\![t\ u]\!]\,\vec{A}\,\vec{a} \triangleq \phi_{B,C}\,([\![t]\!]\,\vec{A}\,\vec{a})\,([\![u]\!]\,\vec{A}\,\vec{a}) \qquad (\text{ただし } B, C \text{ は } [\![t]\!]\,\vec{A}\,\vec{a} \in \mathbf{D}_{B\to C}, [\![u]\!]\,\vec{A}\,\vec{a} \in \mathbf{D}_B \text{ を満たす})$$
$$[\![\Lambda Y.\, v]\!]\,\vec{A}\,\vec{a} \triangleq \Psi_F\,(B \mapsto [\![v]\!]\,\vec{A}[B/Y]\,\vec{a}) \qquad (\text{ただし } F(B) \text{ は } [\![v]\!]\,\vec{A}[B/Y]\,\vec{a} \in \mathbf{D}_{F(B)} \text{ を満たす})$$
$$[\![t_U]\!]\,\vec{A}\,\vec{a} \triangleq \Phi_F\,([\![t]\!]\,\vec{A}\,\vec{a})\,([\![U]\!]\,\vec{A}) \qquad (\text{ただし } F \text{ は } [\![t]\!]\,\vec{A}\,\vec{a} \in \mathbf{D}_{\forall F} \text{ を満たす})$$

[9] ここでの「値」というのは項に割り当てられる予定の数学的対象のことです．型値とは別個のものです．

[10] 「環境」は型環境と別個のものです．また，「環境対」は造語です．

[11] 3 番目の等式において，() 内の条件を満たす B, C が存在しない場合は未定義としておいていいです．5 番目の等式において，() 内の条件を満たす F が存在しない場合も未定義としておいていいです．

$\vec{a}[b/y]$ はやはり $\vec{a}$ を拡張したもので，

$$\vec{a}[b/y]\ [\![x]\!] \triangleq \begin{cases} b & (x = y) \\ \vec{a}\ [\![x]\!] & (x \neq y) \end{cases}$$

と定義されます．

以上の制約を満たす $\mathbf{U}, \to, \forall, \mathbf{D}, \phi, \psi, \Phi, \Psi$ から成る構造を**フレーム** (frame) といいます．さらに，フレームにおいて，$\vec{X}; \vec{x} \vdash t : T$ である任意の $\vec{X}, \vec{x}, t, T$，そして $\vec{X}, \vec{x}$ にかんする任意の環境対 $\langle \vec{A}, \vec{a} \rangle$ について，常に $[\![t]\!]\ \vec{A}\ \vec{a}$ が定義可能ならば，[12] そのフレームは**環境モデル** (environment model) であるといいます．ここで，ある環境モデルを考えましょう．この環境モデルにおいて，項 t と型 T があり，$\vec{X}, \vec{x}$ にかんする任意の環境対 $\langle \vec{A}, \vec{a} \rangle$ が $[\![t]\!]\ \vec{A}\ \vec{a} \in \mathbf{D}_{[\![T]\!]\ \vec{A}}$ を満たしているとき，

$$\vec{X}; \vec{x} \models t : T$$

と書きます．[13]

さて，意味論はしっかり構築できました．意味論を構文論と対応付けましょう．一般に，**健全性** (soundness) というのは，構文論において成立していることが意味論においても成立している，ということです．これから，型についての**健全性**が成立していることを証明していきます．

定理（型の健全性）. 任意の $\vec{X}, \vec{x}, t, T$ について，$\vec{X}; \vec{x} \vdash t : T$ ならば $\vec{X}; \vec{x} \models t : T$ である．

証明. 項 t についての構造的帰納法により証明します．帰納法の仮定は

任意の$\vec{X'}, \vec{x'}, T'$と，t の部分をなす任意の項 t'について，$\vec{X'}; \vec{x'} \vdash t' : T'$ならば$\vec{X'}; \vec{x'} \models t' : T'$

です．

i. $t = y$ のとき
   $\vec{X}; \vec{x} \vdash y : T$ と仮定します．このとき (Var-Trivial) より，$\langle y, T \rangle \in \vec{x}$ です．
   よって，$\vec{X}, \vec{x}$ にかんする任意の環境対 $\vec{A}, \vec{a}$ について，

$$[\![x]\!]\ \vec{A}\ \vec{a} = \vec{a}\ [\![y]\!] \in \mathbf{D}_{[\![T]\!]\ \vec{A}}$$

   であるので，$\vec{X}; \vec{x} \models y : Y$ が成立します．

ii. $t = \lambda\, y : U.\, v$ のとき
   $\vec{X}; \vec{x} \vdash (\lambda\ y : U.\ v) : T$ と仮定します．このとき (Func-Intro) より，ある型 V が存在し，$\vec{X}; \vec{x}, \langle y : U \rangle \vdash v : V$ かつ $\mathrm{Fv}(U) \subset \vec{X}$ かつ $T = U \to V$ です．更に帰納法の仮定より $\vec{X}; \vec{x}, \langle y : U \rangle \models v : V$ となります．
   ここで，$\vec{X}, \vec{x}$ にかんする任意の環境対 $\langle \vec{A}, \vec{a} \rangle$ と $b \in \mathbf{D}_{[\![U]\!]\ \vec{A}}$ を考えます．
   このとき，

$$\begin{aligned} \vec{a}[b/y]\ [\![y]\!] &= b \in \mathbf{D}_{[\![U]\!]\vec{A}} \\ \vec{a}[b/y]\ [\![x_j]\!] &= \vec{a}\ [\![x_j]\!] \in \mathbf{D}_{[\![T_j]\!]\ \vec{A}} \quad (\langle x_j, T_j \rangle \in \vec{x} \text{ かつ } x_j \neq y) \end{aligned}$$

[12] 言い換えれば「$(\mathbf{U} \to \mathbf{U}), (\mathbf{D}_A \to \mathbf{D}_B), (\forall A : \mathbf{U}.\mathbf{D}_{F(A)})$ が System F で表現できる型や項を包み込むのに十分大きければ」ということです．

[13] $\models$ はダブルターンスタイル (double turnstile) などと呼ばれます．TEX では `\vDash` で $\vDash$ が，`\models` で $\models$ (この記事で使っている側) が表示されます．

であるので，$\langle \vec{A}, \vec{a}[b/y] \rangle$ は $\vec{X}$ と $\vec{x}, \langle y : U \rangle$ にかんする環境対です．よって，$[\![v]\!]\ \vec{A}\ \vec{a}[b/y] \in \mathbf{D}_{[\![V]\!]\ \vec{A}}$ であるので，

$$[\![\lambda\, y : U.\ v]\!]\ \vec{A}\ \vec{a} = \psi_{[\![U]\!]\ \vec{A}, [\![V]\!]\ \vec{A}}\ (b \mapsto [\![v]\!]\ \vec{A}\ \vec{a}[b/y]) \in \mathbf{D}_{[\![U]\!]\ \vec{A} \to [\![V]\!]\ \vec{A}} = \mathbf{D}_{[\![U \to V]\!]\ \vec{A}}$$

です．

ゆえに，$\vec{X}; \vec{x} \models (\lambda\, y : U.\ v) : U \to V$ が成立します．

iii. $t = u\ v$ のとき

$\vec{X}; \vec{x} \vdash (u\ v) : T$ と仮定します．このとき (Func-Elim) より，ある型 V が存在し，$\vec{X}; \vec{x} \vdash u : V \to T$ かつ $\vec{X}; \vec{x} \vdash v : V$ です．更に帰納法の仮定より $\vec{X}; \vec{x} \models u : V \to T$ かつ $\vec{X}; \vec{x} \models v : V$ です．

よって $\vec{X}; \vec{x}$ にかんする任意の環境対 $\langle \vec{A}, \vec{a} \rangle$ について，

$B = [\![V]\!]\ \vec{A}, C = [\![T]\!]\ \vec{A}$ とすると，

$$[\![u]\!]\ \vec{A}\ \vec{a} \in \mathbf{D}_{[\![V \to T]\!]\ \vec{A}} = \mathbf{D}_{B \to C} \quad \text{かつ} \quad [\![v]\!]\ \vec{A}\ \vec{a} \in \mathbf{D}_{[\![V]\!]\ \vec{A}} = \mathbf{D}_B$$

であるので，

$$[\![u\ v]\!]\ \vec{A}\ \vec{a} = \phi_{B,C}\ ([\![u]\!]\ \vec{A}\ \vec{a})\ ([\![v]\!]\ \vec{A}\ \vec{a}) \in \mathbf{D}_B = \mathbf{D}_{[\![V]\!]\ \vec{A}}$$

です．

ゆえに，$\vec{X}; \vec{x} \models (u\ v) : T$ が成立します．

iv. $t = \Lambda Y.\ v$ のとき

$\vec{X}; \vec{x} \vdash (\Lambda Y.\ v) : T$ と仮定します．このとき (Forall-Intro) より，ある型 V が存在し，$\vec{X}, Y; \vec{x} \vdash v : V$ かつ $T = \forall Y.\ V$ です．更に帰納法の仮定より，$\vec{X}, Y; \vec{x} \models v : V$ です．

ここで，$\vec{X}, \vec{x}$ にかんする任意の環境対 $\langle \vec{A}, \vec{a} \rangle$ について，

$\langle \vec{A}[B/Y], \vec{a} \rangle$ は $\vec{X}, Y; \vec{x}$ にかんする環境対であるので，$[\![v]\!]\ \vec{A}[B/Y]\ \vec{a} \in \mathbf{D}_{[\![V]\!]\ \vec{A}[B/Y]}$ です．

よって，$F(B)$ を $[\![V]\!]\ \vec{A}[B/Y] \in \mathbf{D}_C$ を満たす C とすると，

$$[\![\Lambda Y.\ v]\!]\ \vec{A}\ \vec{a} = \Psi_F(B \mapsto [\![v]\!]\ \vec{A}[B/Y]\ \vec{a}) \in \mathbf{D}_{\forall F} = \mathbf{D}_{[\![\forall Y.\ V]\!]\ \vec{A}}$$

です．

ゆえに，$\vec{X}; \vec{x} \models (\Lambda Y.\ v) : \forall Y.\ V$ が成立します．

v. $t = v_U$ のとき

$\vec{X}; \vec{x} \vdash v_U$ と仮定します．このとき (Forall-Elim) より，ある型 V が存在し，$\vec{X}; \vec{x} \vdash v : \forall Y.\ V$ かつ $T = V[U/Y]$ です．更に帰納法の仮定より，$\vec{X}; \vec{x} \vdash v : \forall Y.\ V$ です．よって，$\vec{X}, \vec{x}$ にかんする任意の環境対 $\langle \vec{A}, \vec{a} \rangle$ について，$F(B) = [\![V]\!]\ \vec{A}[B/Y]$ とすると，

$$[\![v]\!]\ \vec{A}\ \vec{a} \in \mathbf{D}_{[\![\forall Y.\ V]\!]\ \vec{A}} = \mathbf{D}_{\forall F}$$

であるので，

$$[\![v_U]\!]\ \vec{A}\ \vec{a} = \Phi_F\ ([\![v]\!]\ \vec{A}\ \vec{a}) \in \mathbf{D}_{F([\![U]\!]\ \vec{A})}$$

です．更に，

$$F([\![U]\!]\ \vec{A}) = [\![V]\!]\ \vec{A}\ [[\![U]\!]\ \vec{A}/Y] = [\![V[U/Y]]\!]\ \vec{A}$$

であるので，

$$[\![v_U]\!]\ \vec{A}\ \vec{a} \in \mathbf{D}_{[\![V[U/Y]]\!]\ \vec{A}}$$

です．

ゆえに，$\vec{X}; \vec{x} \models v_U : V[U/Y]$ が成立します．

以上により t についての構造的帰納法が完成しました． (Q.E.D.)

5 二項関係で遊ぼう

2 つの集合 A, B の間の**二項関係** (binary relation) $\mathcal{A} : A \leftrightarrow B$ とは，直積 $A \times B$ の部分集合のことです．[14] 二項関係は $\mathcal{A}, \mathcal{B}, \mathcal{C}$ で表します．また，A, B の間の二項関係全体の集合は $A \leftrightarrow B$ と書きます．つまり，$\mathcal{A} : A \leftrightarrow B$ は $\mathcal{A} \in A \leftrightarrow B$ と同じことです．A, B の間の二項関係は「**二引数の述語**」として捉えることもできます．つまり，$A \leftrightarrow B$ の二項関係は A と B から真偽値集合 $\mathbb{B} = \{\mathrm{True}, \mathrm{False}\}$ への二引数関数 $A \times B \to \mathbb{B}$ と対応しているのです．

二項関係というと難しそうですが，身近にいろいろあります．たとえば，「$n \leq m$」とか「n と m は互いに素」というのは，$\mathbb{N} \leftrightarrow \mathbb{N}$ の二項関係です（$\mathbb{N}$ は自然数全体の集合です）．また，人全体の集合を H，プログラミング言語の集合を L として，「h は l を使える」というのは，$H \leftrightarrow L$ の二項関係です．集合の世界で捉えると，「条件を満たす組の集合」として考えるのが自然なのです．

関数 $f : A \to B$ も A, B 間の二項関係とみなすことができます．

$$\langle x, y \rangle \in f \iff y = f(x)$$

ということです．別の見方をすると，二項関係 $f : A \leftrightarrow B$ が A から B への関数であるとは，

$$\text{任意の } x \in A \text{ について } \langle x, y \rangle \in f \text{ となる } y \in B \text{ が一意に存在する}$$

ということです．「一意に存在する」というのは「ちょうど 1 つだけ存在する」ということです．この記事では，関数は二項関係の一種であるとみなします．また，関数は f, g, h, k で表すことにします．

二項関係は関数に比べて対称性が高いのが特徴です．任意の二項関係 $\mathcal{A} : A \leftrightarrow B$ について，**逆関係** (inverse relation) $\mathcal{A}^{-1} : B \leftrightarrow A$ は

$$\mathcal{A}^{-1} \triangleq \{\langle y, x \rangle \mid \langle x, y \rangle \in \mathcal{A}\}$$

と定義されます．逆関係の逆関係 $(\mathcal{A}^{-1})^{-1}$ は元の関係 $\mathcal{A}$ と同じですね．関数 f の逆関係 f^{-1} は，f が全単射であるとき，かつそのときに限り，関数となり，**逆関数** (inverse function) と呼ばれます．関数は逆関係を取ると関数でなくなる可能性がありますから，この意味で二項関係と比べて対称性が低いといえます．

二項関係 $\mathcal{A} : A \leftrightarrow B, \mathcal{B} : B \leftrightarrow C$ の**合成** (composite) $\mathcal{B} \circ \mathcal{A} : A \leftrightarrow C$ は

$$\mathcal{B} \circ \mathcal{A} \triangleq \{\langle x, z \rangle \mid \langle x, y \rangle \in \mathcal{A} \text{ かつ } \langle y, z \rangle \in \mathcal{B} \text{ となる } y \in B \text{ が存在する } \}$$

と定義されます．関数 f, g の合成 $g \circ f$ も二項関係の合成とみなせます．

さて，ここでは，System F において二項関係を活用します．二項関係を元にしたもうひとつの意味論をつくりだすのです．具体的にいうと，**型を二項関係とみなします**．かなり不思議かもしれませんが，意味論の世界は**自由**なのです！とにかく，何をやっているか見てください．そしてこの意味論のすごさを実感してください．

[14] $A \leftrightarrow B$ というのはこの記事での臨時的な書き方です．標準的な書き方は特に無いようです．ちなみに論理的同値を表す場合，この記事では $\iff$ を使います．

型変数の集合 $\vec{X}$ に対する**関係環境** (relation environement) $\vec{\mathcal{A}}$ とは，任意の $X_i \in \vec{X}$ を二項関係 $\vec{\mathcal{A}}\,[\![X]\!]$ に対応付けるもののことです．さらに，$\vec{X}$ に対する型環境 $\vec{A}, \vec{A'}$ があり，任意の $X_i \in \vec{X}$ について対応付けられた二項関係が $\vec{\mathcal{A}}\,[\![X_i]\!] : \mathbf{D}_{\vec{A}\,[\![X_i]\!]} \leftrightarrow \mathbf{D}_{\vec{A'}\,[\![X_i]\!]}$ を満たすとき，$\vec{\mathcal{A}} : \vec{A} \Leftrightarrow \vec{A'}$ と書きます．

これまで，前節の意味論において型や項を**型値や値として解釈**してきました．ここでは，型を**二項関係として解釈**したいです．そこで，二項関係についての演算を 2 つだけ定義しておきます．とっても単純です．

1 つ目．二項関係 $\mathcal{A} : \mathbf{D}_A \leftrightarrow \mathbf{D}_{A'}, \mathcal{B} : \mathbf{D}_B \leftrightarrow \mathbf{D}_{B'}$ からできる二項関係 $\mathcal{A} \to \mathcal{B} : \mathbf{D}_{A\to B} \leftrightarrow \mathbf{D}_{A'\to B'}$ を

$$\mathcal{A} \to \mathcal{B} \triangleq \{\langle c, c'\rangle \mid \langle a, a'\rangle \in \mathcal{A} \text{ ならば } \langle \phi_{A,B}\ c\ a, \phi_{A',B'}\ c'\ a'\rangle \in \mathcal{B}\}$$

と定義します．なお，$(\mathcal{A} \to \mathcal{B})^{-1} = \mathcal{A}^{-1} \to \mathcal{B}^{-1}$ が成立することにも注目してください．

2 つ目．関数 $F : \mathbf{U} \to \mathbf{U}, F' : \mathbf{U} \to \mathbf{U}$ を考えます．$\mathcal{F}$ を二項関係 $\mathcal{A} : \mathbf{D}_A \leftrightarrow \mathbf{D}_{A'}$ を取って二項関係 $\mathcal{F}(\mathcal{A}) : \mathbf{D}_{F(A)} \leftrightarrow \mathbf{D}_{F'(A')}$ を返す関数とします．このとき，二項関係 $\forall\mathcal{F} : \mathbf{D}_{\forall F} \leftrightarrow \mathbf{D}_{\forall F'}$ を

$$\forall\mathcal{F} \triangleq \{\langle d, d'\rangle \mid \text{任意の } A, A', \mathcal{A} : \mathbf{D}_A \leftrightarrow \mathbf{D}_{A'} \text{ について } \langle \Phi_F\ d\ A, \Phi_{F'}\ d'\ A'\rangle \in \mathcal{F}(\mathcal{A})\}$$

と定義します．

さて，型 T の自由型変数をすべて含む集合 $\vec{X}$ に対する関係環境 $\vec{\mathcal{A}}$ があるとき，T の $\vec{\mathcal{A}}$ における**解釈**が次のようにして定義されます．

$$\begin{aligned}
[\![X]\!]\ \vec{\mathcal{A}} &\triangleq \vec{\mathcal{A}}\,[\![X]\!] \\
[\![U \to V]\!]\ \vec{\mathcal{A}} &\triangleq [\![U]\!]\ \vec{\mathcal{A}} \to [\![V]\!]\ \vec{\mathcal{A}} \\
[\![\forall Y.\ V]\!]\ \vec{\mathcal{A}} &\triangleq \forall(\mathcal{B} \mapsto [\![V]\!]\ \vec{\mathcal{A}}[\mathcal{B}/Y])
\end{aligned}$$

$\vec{\mathcal{A}}[\mathcal{B}/Y]$ はやはり

$$\vec{\mathcal{A}}[\mathcal{B}/Y]\,[\![X]\!] \triangleq \begin{cases} \mathcal{B} & (X = Y) \\ \vec{\mathcal{A}}\,[\![X]\!] & (X \neq Y) \end{cases}$$

により定義されます．

$\langle \vec{A}, \vec{a}\rangle$ と $\langle \vec{A'}, \vec{a'}\rangle$ が $\vec{X}, \vec{x}$ にかんする環境対であり，二項関係 $\vec{\mathcal{A}}$ について $\vec{\mathcal{A}} : \vec{A} \Leftrightarrow \vec{A'}$ であり，かつ任意の $\langle x_j, T_j\rangle \in \vec{x}$ について $\langle \vec{a}[\![x_j]\!], \vec{a'}[\![x_j]\!]\rangle \in [\![T_j]\!]\ \vec{\mathcal{A}}$ を満たしているとき，$\langle \vec{\mathcal{A}}, \vec{A}, \vec{a}, \vec{A'}, \vec{a'}\rangle$ は $\vec{X}, \vec{x}$ にかんする**環境五つ組** (environment quintuple) であるといいます．[15] そして，項 t と型 T があり，$\vec{X}, \vec{x}$ にかんする任意の環境五つ組 $\langle \vec{\mathcal{A}}, \vec{A}, \vec{a}, \vec{A'}, \vec{a'}\rangle$ が $\langle [\![t]\!]\ \vec{A}\ \vec{a}, [\![t]\!]\ \vec{A'}\ \vec{a'}\rangle \in [\![T]\!]\ \vec{\mathcal{A}}$ を満たしているとき，

$$\vec{X}; \vec{x} \Vdash t : T$$

と書きます．[16]

定理（パラメトリシティ）． 任意の $\vec{X}, \vec{x}, t, T$ について，$\vec{X}; \vec{x} \vdash t : T$ ならば $\vec{X}; \vec{x} \Vdash t : T$ である．

証明． t についての構造的帰納法で証明します．帰納法の仮定は

任意の$\vec{X'}, \vec{x'}, T'$と，t の部分をなす任意の項 t'について，$\vec{X'}; \vec{x'} \vdash t' : T'$ならば$\vec{X'}; \vec{x'} \Vdash t' : T'$

です．

---

[15] 「環境五つ組」は造語です．

[16] $\Vdash$ は $\models$ の変種です．特に決まった名前はありません．

i. $t = y$ のとき

$\vec{X}; \vec{x} \vdash y : T$ と仮定します．このとき (Var-Trivial) より，$\langle y, T\rangle \in \vec{x}$ です．

よって，$\vec{X}, \vec{x}$ にかんする任意の環境五つ組 $\langle \vec{\mathcal{A}}, \vec{A}, \vec{a}, \vec{A'}, \vec{a}\rangle$ について，

$$\langle [\![y]\!]\ \vec{A}\ \vec{a}, [\![y]\!]\ \vec{A'}\ \vec{a'}\rangle = \langle \vec{a}[\![y]\!], \vec{a'}[\![y]\!]\rangle \in [\![T]\!]\ \vec{\mathcal{A}}$$

であるので，$\vec{X}; \vec{x} \models y : T$ が成立します．

ii. $t = \lambda\, y : U.\ v$ のとき

$\vec{X}; \vec{x} \vdash (\lambda\ y : U.\ v) : T$ と仮定します．このとき (Func-Intro) より，ある型 V が存在し，$\vec{X}; \vec{x}, \langle y : U\rangle \vdash v : V$ かつ $\mathrm{Fv}(U) \subset \vec{X}$ かつ $T = U \to V$ です．さらに帰納法の仮定より $\vec{X}; \vec{x}, \langle y : U\rangle \models v : V$ となります．

ここで，$\vec{X}, \vec{x}$ にかんする任意の環境五つ組 $\langle \vec{\mathcal{A}}, \vec{A}, \vec{a}, \vec{A'}, \vec{a}\rangle$ を考えます．

いま，$\langle b, b'\rangle \in [\![U]\!]\ \vec{\mathcal{A}}$ であると仮定すると，

$b \in \mathbf{D}_{[\![U]\!]\ \vec{A}}, b' \in \mathbf{D}_{[\![U]\!]\vec{A'}}$ であり，

$$\phi_{[\![U]\!]\ \vec{A}, [\![V]\!]\ \vec{A}}\ ([\![\lambda\, y : U.\ v]\!]\ \vec{A}\ \vec{a})\ b = [\![v]\!]\ \vec{A}\ \vec{a}[b/y]$$
$$\phi_{[\![U]\!]\ \vec{A'}, [\![V]\!]\ \vec{A'}}\ ([\![\lambda\, y : U.\ v]\!]\ \vec{A'}\ \vec{a'})\ b' = [\![v]\!]\ \vec{A'}\ \vec{a'}[b'/y]$$

です．$\langle \vec{A}, \vec{a}[b/y]\rangle, \langle \vec{A'}, \vec{a'}[b'/y]\rangle$ はともに $\vec{X}$ と $\vec{x}, \langle y : U\rangle$ にかんする環境対です．[17] さらに，

$$\langle \vec{a}[b/y][\![y]\!], \vec{a'}[b'/y][\![y]\!]\rangle = \langle b, b'\rangle \in [\![U]\!]\vec{\mathcal{A}}$$
$$\langle \vec{a}[b/y][\![x_j]\!], \vec{a'}[b'/y][\![x_j]\!]\rangle = \langle \vec{a}[\![x_j]\!], \vec{a'}[\![x_j]\!]\rangle \in [\![T_j]\!]\ \vec{\mathcal{A}}\ \ (\langle x_j, T_j\rangle \in \vec{X} \text{ かつ } x_j \neq y)$$

であるので，$\langle \vec{\mathcal{A}}, \vec{A}, \vec{a}[b/y], \vec{A'}, \vec{a'}[b'/y]\rangle$ は $\vec{X}; \vec{x}, \langle y : U\rangle$ にかんする環境五つ組です．

よって，

$$\langle [\![v]\!]\ \vec{A}\ \vec{a}[b/y], [\![v]\!]\ \vec{A'}\ \vec{a'}[b'/y]\rangle \in [\![V]\!]\ \vec{\mathcal{A}}$$

です．

よって，$\langle [\![\lambda\, y : U.\ v]\!]\ \vec{A}\ \vec{a}, [\![\lambda\, y : U.\ v]\!]\ \vec{A'}\ \vec{a'}\rangle \in ([\![U]\!]\ \vec{\mathcal{A}} \to [\![V]\!]\ \vec{\mathcal{A}}) = [\![U \to V]\!]\ \vec{\mathcal{A}}$ です．

ゆえに，$\vec{X}; \vec{x} \models (\lambda\, y : U.\ v) : U \to V$ が成立します．

iii. $t = u\ v$ のとき

$\vec{X}; \vec{x} \vdash (u\ v) : T$ と仮定します．このとき (Func-Elim) より，ある型 V が存在し，$\vec{X}; \vec{x} \vdash u : V \to T$ かつ $\vec{X}; \vec{x} \vdash v : V$ です．さらに帰納法の仮定より $\vec{X}; \vec{x} \models u : V \to T$ かつ $\vec{X}; \vec{x} \models v : V$ です．

ここで，$\vec{X}, \vec{x}$ にかんする任意の環境五つ組 $\langle \vec{\mathcal{A}}, \vec{A}, \vec{a}, \vec{A'}, \vec{a}\rangle$ を考えます．

すると，

$$\langle [\![v]\!]\ \vec{A}\ \vec{a}, [\![v]\!]\ \vec{A'}\ \vec{a'}\rangle \in [\![V]\!]\ \vec{\mathcal{A}}$$
$$\langle [\![u]\!]\ \vec{A}\ \vec{a}, [\![u]\!]\ \vec{A'}\ \vec{a'}\rangle \in [\![V \to T]\!]\ \vec{\mathcal{A}} = [\![V]\!]\ \vec{\mathcal{A}} \to [\![T]\!]\ \vec{\mathcal{A}}$$

より，

$$\langle [\![u\ v]\!]\ \vec{A}\ \vec{a}, [\![u\ v]\!]\ \vec{A'}\ \vec{a'}\rangle = \langle \phi\ ([\![u]\!]\ \vec{A}\ \vec{a})\ ([\![v]\!]\ \vec{A}\ \vec{a}), \phi\ ([\![u]\!]\ \vec{A'}\ \vec{a'})\ ([\![v]\!]\ \vec{A'}\ \vec{a'})\rangle \in [\![T]\!]\ \vec{\mathcal{A}}$$

です．

[17] 命題 1 の証明の中で同様のことを証明しました．

ゆえに $\vec{X};\vec{x} \models (u\ v) : T$ が成立します．

iv. $t = \Lambda Y.\ v$ のとき

$\vec{X};\vec{x} \vdash (\Lambda Y.\ v) : T$ と仮定します．このとき (Forall-Intro) より，ある型 V が存在し，$\vec{X}, Y;\vec{x} \vdash v : V$ かつ $T = \forall Y.\ V$ です．さらに帰納法の仮定より，$\vec{X}, Y;\vec{x} \models v : V$ です．

ここで，$\vec{X},\vec{x}$ にかんする任意の環境五つ組 $\langle \vec{\mathcal{A}}, \vec{A}, \vec{a}, \vec{A'}, \vec{a} \rangle$ を考えます．

$F(B)$ を $[\![V]\!]\ \vec{A}[B/Y] \in \mathbf{D}_C$ を満たす C とし，$F'(B)$ を $[\![V]\!]\ \vec{A'}[B'/Y] \in \mathbf{D'}_C$ を満たす C' とし，$\mathcal{F}(\mathcal{B}) = [\![V]\!]\ \vec{\mathcal{A}}[\mathcal{B}/Y]$ とします．

任意の B, B' と $\mathcal{B} : B \Leftrightarrow B'$ について，

$\langle \vec{A}[B/Y], \vec{a} \rangle$ も $\langle \vec{A'}[B'/Y], \vec{a'} \rangle$ も $\vec{X}, Y$ と $\vec{x}$ にかんする環境対です．ここで，

$$\vec{\mathcal{A}}[\mathcal{B}/Y]\ [\![Y]\!] = \mathcal{B} \in (\mathbf{D}_B \leftrightarrow \mathbf{D}_{B'}) = (\mathbf{D}_{\vec{A}[B/Y]\ [\![Y]\!]} \leftrightarrow \mathbf{D}_{\vec{A'}[B'/Y]\ [\![Y]\!]})$$

であり，かつ $X_i \in X$ かつ $X_i \neq Y$ のとき

$$\vec{\mathcal{A}}[\mathcal{B}/Y]\ [\![X_i]\!] \in (\mathbf{D}_{\vec{A}[\![X_i]\!]} \leftrightarrow \mathbf{D}_{\vec{A'}\ [\![X_i]\!]}) = (\mathbf{D}_{\vec{A}[B/Y]\ [\![X_i]\!]} \leftrightarrow \mathbf{D}_{\vec{A'}[B'/Y]\ [\![X_i]\!]})$$

です．また，任意の $\langle x_j, T_j \rangle \in \vec{x}$ について，T_j は自由型変数 Y を含みませんから

$$\langle \vec{a}\ [\![x_j]\!], \vec{a'}\ [\![x_j]\!] \rangle \in \vec{\mathcal{A}}\ [\![T_j]\!] = \vec{\mathcal{A}}[\mathcal{B}/Y]\ [\![T_j]\!]$$

です．よって，$\langle \vec{\mathcal{A}}[\mathcal{B}/Y], \vec{A}[B/Y], \vec{a}, \vec{A'}[B'/Y], \vec{a'} \rangle$ は $\vec{X}, Y$ と $\vec{x}$ にかんする環境五つ組であるので，

$$\langle \Phi_F\ ([\![\Lambda Y.\ v]\!]\ \vec{A}\ \vec{a})\ B, \Phi_{F'}\ ([\![\Lambda Y.\ v]\!]\ \vec{A'}\ \vec{a'})\ B' \rangle = \langle [\![v]\!]\ \vec{A}[B/Y]\ \vec{a}, [\![v]\!]\ \vec{A'}[B'/Y]\ \vec{a'} \rangle \in [\![V]\!]\ \vec{\mathcal{A}}[\mathcal{B}/Y]$$

です．

よって，

$$\langle [\![\Lambda Y.\ v]\!]\ \vec{A}\ \vec{a}, [\![\Lambda Y.\ v]\!]\ \vec{A'}\ \vec{a'})\ B' \rangle \in \forall \mathcal{F} = [\![\forall Y.\ V]\!]\ \mathcal{A}$$

です．

ゆえに $\vec{X};\vec{x} \models (\Lambda Y.\ v) : \forall Y.\ V$ が成立します．

v. $t = v_U$ のとき

$\vec{X};\vec{x} \vdash v_U : T$ と仮定します．このとき (Forall-Elim) より，ある型 V が存在し，$\vec{X};\vec{x} \vdash v : \forall Y.\ V$ かつ $T = V[U/Y]$ です．さらに帰納法の仮定より，$\vec{X};\vec{x} \models v : \forall Y.\ V$ です．

ここで，$\vec{X},\vec{x}$ にかんする任意の環境五つ組 $\langle \vec{\mathcal{A}}, \vec{A}, \vec{a}, \vec{A'}, \vec{a} \rangle$ を考えます．

このとき，

$$\langle [\![v]\!]\ \vec{A}\ \vec{a}, [\![v]\!]\ \vec{A'}\ \vec{a'} \rangle \in [\![\forall Y.\ V]\!]\ \vec{\mathcal{A}} = \forall (\mathcal{B} \mapsto [\![V]\!]\ \vec{\mathcal{A}}[\mathcal{B}/Y])$$

です．$F(B) = [\![V]\!]\ \vec{A}[B/Y]$ とし，$F'(B') = [\![V]\!]\ \vec{A'}[B'/Y]$ とします．

$$[\![U]\!]\ \vec{\mathcal{A}} : \mathbf{D}_{[\![U]\!]\ \vec{A}} \leftrightarrow \mathbf{D}_{[\![U]\!]\ \vec{A'}}$$

であるので，

$$\langle \Phi_F\ ([\![v]\!]\ \vec{A}\ \vec{a})\ ([\![U]\!]\ \vec{A}), \Phi_{F'}\ ([\![v]\!]\ \vec{A'}\ \vec{a'})\ ([\![U]\!]\ \vec{A'}) \rangle \in [\![V]\!]\ \vec{\mathcal{A}}[[\![U]\!]\ \vec{\mathcal{A}}/Y]$$

です．さらに，

$$\begin{aligned}\llbracket v_U \rrbracket\ \vec{A}\ \vec{a} &= \Phi_F\ (\llbracket v \rrbracket\ \vec{A}\ \vec{a})\ (\llbracket U \rrbracket\ \vec{A}) \\ \llbracket v_U \rrbracket\ \vec{A'}\ \vec{a'} &= \Phi_{F'}\ (\llbracket v \rrbracket\ \vec{A'}\ \vec{a'})\ (\llbracket U \rrbracket\ \vec{A'}) \\ \llbracket V[U/Y] \rrbracket\ \vec{\mathcal{A}} &= \llbracket V \rrbracket\ \vec{\mathcal{A}}[\llbracket U \rrbracket\ \vec{\mathcal{A}}/Y]\end{aligned}$$

であるので，

$$\langle \llbracket v_U \rrbracket\ \vec{A}\ \vec{a}, \llbracket v_U \rrbracket\ \vec{A'}\ \vec{a'} \rangle \in \llbracket V \rrbracket\ \vec{\mathcal{A}}[\llbracket U \rrbracket\ \vec{\mathcal{A}}/Y]$$

です．

ゆえに $\vec{X}; \vec{x} \models v_U : V[U/Y]$ が成立します．

以上により t についての構造的帰納法が完成しました． (Q.E.D.)

6 パラメトリシティを活用しよう

たった今証明したパラメトリシティという定理はかなり一般性を持っています．しかし，そのままだとどう使ってよいか露頭に迷ってしまいます．そこで，パラメトリシティから**自然性** (naturalness) と呼ばれる性質を導いて行くことにします．

まず，$\mathcal{A} : \mathbf{D}_{\llbracket T \rrbracket\ \varnothing} \leftrightarrow \mathbf{D}_{\llbracket U \rrbracket\ \varnothing}$ と，$\varnothing; \varnothing \vdash v : T \to U$ を満たす項 v が

$$\begin{aligned}&\varnothing; \varnothing \vdash t : T \text{ を満たす項 } t \text{ と } \varnothing; \varnothing \vdash t' : T \text{ を満たす項 } t' \text{ について} \\ &\qquad 「\langle \llbracket t \rrbracket\ \varnothing\ \varnothing, \llbracket t' \rrbracket\ \varnothing\ \varnothing \rangle \in \mathcal{A}」 \iff 「t' \equiv v\ t」\end{aligned}$$

を満たしているとき，

$$\llbracket v \rrbracket \hookrightarrow \mathcal{A}$$

と書くことにします．[18] なお，$\mathcal{A} = \phi_{\llbracket T \rrbracket\ \varnothing, \llbracket U \rrbracket\ \varnothing}(\llbracket v \rrbracket\ \varnothing\ \varnothing)$ ならば $\llbracket v \rrbracket \hookrightarrow \mathcal{A}$ ですが，その逆は一般には成立しません．

関数型と全称量化型のそれぞれについて次の補題が成立しています．

補題（関数型のトリック）． $\mathcal{A} : \mathbf{D}_{\llbracket U \rrbracket\ \varnothing} \leftrightarrow \mathbf{D}_{\llbracket V \rrbracket\ \varnothing}$ と $\mathcal{B} : \mathbf{D}_{\llbracket U' \rrbracket\ \varnothing} \leftrightarrow \mathbf{D}_{\llbracket V' \rrbracket\ \varnothing}$ があり，$\llbracket v \rrbracket \hookrightarrow \mathcal{A}^{-1}$ かつ $\llbracket v' \rrbracket \hookrightarrow \mathcal{B}$ であるする．このとき，$\llbracket \lambda\, y : U \to U'.\, \lambda\, x : V.\, v'\ (y\ (v\ x)) \rrbracket \hookrightarrow (\mathcal{A} \to \mathcal{B})$ である．

証明．

$$\begin{aligned}\langle \llbracket t \rrbracket\ \varnothing\ \varnothing, \llbracket t' \rrbracket\ \varnothing\ \varnothing \rangle \in \mathcal{A} \to \mathcal{B} &\iff \langle \llbracket u \rrbracket\ \varnothing\ \varnothing, \llbracket u' \rrbracket\ \varnothing\ \varnothing \rangle \in \mathcal{A} \text{ ならば } \langle \llbracket t\ u \rrbracket\ \varnothing\ \varnothing, \llbracket t'\ u' \rrbracket\ \varnothing\ \varnothing \rangle \in \mathcal{B} \\ &\iff \text{任意の } u' \text{ について } v'\ (t\ (v\ u')) \equiv t'\ u' \\ &\iff t' \equiv \lambda\, x : V.\, v'\ (t\ (v\ x)) \\ &\iff t' \equiv (\lambda\, y : U \to U'.\, \lambda\, x : V.\, v'\ (y\ (v\ x)))\ t\end{aligned}$$

(Q.E.D.)

補題（全称量化型のトリック）． 型 U と，U の自由型変数をすべて含む $\vec{X}$ と，それに対する型環境 $\vec{A}, \vec{A'}$ と，関係環境 $\vec{\mathcal{A}} : \vec{A} \Leftrightarrow \vec{A'}$ があるとする．

また，任意の B について $\llbracket V \rrbracket\ [B/X] = \llbracket U \rrbracket\ \vec{A}[B/X]$ である型 V と，任意の B' について $\llbracket V' \rrbracket\ [B'/X] = \llbracket U \rrbracket\ \vec{A'}[B'/X]$ である型 V' があるとする．

[18] これは臨時につくった記号です．

そして，関数 $\mathcal{F}(\mathcal{B}) = [\![U]\!]\,\vec{\mathcal{A}}[\mathcal{B}/X]$ があるとする．
さらに，任意の閉じた型 T について，$[\![v_T]\!] \hookrightarrow \mathcal{F}(\mathrm{id}_{[\![T]\!]}\,\varnothing)$ であるとする．
このとき，$[\![\lambda\, y : \forall X.\, V.\, \Lambda X.\, v_X\; y]\!] \hookrightarrow \forall\mathcal{F}$ である．

証明． $\varnothing;\varnothing \vdash t : \forall X.\, V$ かつ $\varnothing;\varnothing \vdash t' : \forall X.\, V'$ であるとき，

$$\begin{aligned}\langle [\![t]\!]\,\varnothing\,\varnothing, [\![t']\!]\,\varnothing\,\varnothing\rangle \in \forall\mathcal{F} &\Longrightarrow \text{任意の閉じた型 } T \text{ について } \langle [\![t_T]\!]\,\varnothing\,\varnothing, [\![t'_T]\!]\,\varnothing\,\varnothing\rangle \in \mathcal{F}(\mathrm{id}_{[\![T]\!]}\,\varnothing)\\ &\Longrightarrow \text{任意の閉じた型 } T \text{ について } v_T\; t_T \equiv t'_T\\ &\Longrightarrow t' \equiv \Lambda X.\, v_X\; t_X\\ &\Longrightarrow t' \equiv (\lambda\, y : (\forall X.\, V).\, \Lambda X.\, v_X\; y_X)\; t\end{aligned}$$

が成立しています．そして，$\langle [\![t]\!]\,\varnothing\,\varnothing, [\![(\lambda\, y : (\forall X.\, V).\, \Lambda X.\, v_X\; y_X)\; t]\!]\,\varnothing\,\varnothing\rangle \in \forall\mathcal{F}$ であるので，[19] 補題は成立します． (Q.E.D.)

次は型について少し工夫します．ここでつくった System F には Haskell のような型引数を受け取る型（Maybe や Either などの，類が $*$ でなく $* \to *$ や $* \to * \to *$ などである型）が構文的に用意されていませんでした．ここでは糖衣構文的に型引数を受け取る型を考えることにします．たとえば，

$$Q(X) \triangleq \forall Y.\, (X \to Y) \to Y$$

ということです．

型について，**正の位置**と**負の位置**という概念を次のように定義します．

- X は，X において正の位置に出現する．
- X は，U において正の位置に出現するとき，$T \to U$ において正の位置に出現する．
- X は，U において負の位置に出現するとき，$T \to U$ において負の位置に出現する．
- X は，T において正の位置に出現するとき，$T \to U$ において負の位置に出現する．
- X は，T において負の位置に出現するとき，$T \to U$ において正の位置に出現する．
- X は，T において正の位置に出現するとき，$\forall Y.\, T$ において正の位置に出現する．
- X は，T において負の位置に出現するとき，$\forall Y.\, T$ において負の位置に出現する．
- X は，$\forall X.\, T$ において出現しない．

まあ簡単な話です．例えば $Q(X) \triangleq \forall Y.\, (X \to Y) \to Y$ とすると，$Q(X)$ において X は正の位置にのみ出現します．関数型の引数側において符号が反転するというだけのことです．

さていま，$P(X)$ において X が正の位置にのみ出現するとします．$\varnothing;\varnothing \vdash u : T \to U$ である項 u があるとき，

$$[\![v]\!] \hookrightarrow [\![P(X)]\!]\,[[\![u]\!]\,\varnothing\,\varnothing/X]$$

を満たす v を $P(u)$ と書くことにします．いろいろな P について，**関数型のトリック**と**全称量化型のトリック**から，$P(u)$ を具体的に求めてみてください．

これは実は**関手**になっています．つまり，一般に

$$P(\mathrm{id}_T) \equiv \mathrm{id}_{P(T)} \qquad \text{(SystemF-Functor-Id)}$$
$$P(u \circ u') \equiv P(u) \circ P(u') \qquad \text{(SystemF-Functor-Comp)}$$

[19] これは正しいと思うのですが，これを書いた時点で僕には証明できませんでした．おそらくパラメトリシティを上手く使えば良いと思います．

が成り立つということです．ぜひ確かめてみてください．

なお

$$\mathrm{id} : \forall X.\, X \to X$$
$$\mathrm{id} \triangleq \Lambda X.\, \lambda\, x : X.\, x$$

と定義します．また，$t : T \to U, u : U \to V$ について，

$$u \circ t \triangleq \lambda\, x : T.\, u\ (t\ x)$$

とします（型に関係なく $\circ$ と書くことにします）．

さて，準備は整いました．次の定理が証明できます．

定理（自然性）． $\varnothing; \varnothing \vdash t : \forall X.\, P(X) \to Q(X)$ かつ $\varnothing; \varnothing \vdash u : T \to U$ であるとき，$t_T \circ P(u) \equiv Q(u) \circ t_U$ である．

証明． 項 t についてのパラメトリシティより，任意の $A, A', \mathcal{A} : \mathbf{D}_A \leftrightarrow \mathbf{D}_{A'}$ について

$$\langle [\![t_X]\!]\ [A/X]\ \varnothing, [\![t_X]\!]\ [A'/X]\ \varnothing \rangle \in [\![P(X) \to Q(X)]\!]\ [\mathcal{A}/X] = [\![P(X)]\!]\ [\mathcal{A}/X] \to [\![Q(X)]\!]\ [\mathcal{A}/X]$$

が成立しています．上の式に $A = [\![T]\!]\ \varnothing, A' = [\![U]\!]\ \varnothing, \mathcal{A} = [\![u]\!]\ \varnothing\ \varnothing$ を代入して整理すると，

$$\langle [\![t_T]\!]\ \varnothing\ \varnothing, [\![t_U]\!]\ \varnothing\ \varnothing \rangle \in [\![P(X)]\!]\ [[\![u]\!]\ \varnothing\ \varnothing/X] \to [\![Q(X)]\!]\ [[\![u]\!]\ \varnothing\ \varnothing/X]$$
$$\therefore \langle [\![v]\!]\ \varnothing\ \varnothing, [\![v']\!]\ \varnothing\ \varnothing \rangle \in [\![P(X)]\!]\ [[\![u]\!]\ \varnothing\ \varnothing/X] \text{ ならば } \langle [\![t_T\ v]\!]\ \varnothing\ \varnothing, [\![t_U\ v']\!]\ \varnothing\ \varnothing \rangle \in [\![Q(X)]\!]\ [[\![u]\!]\ \varnothing\ \varnothing/X]$$
$$\therefore \text{任意の } v' \text{ について } t_T\ (P(u)\ v') \equiv Q(u)\ (t_U\ v')$$
$$\therefore t_T \circ P(u) \equiv Q(u) \circ t_U$$

(Q.E.D.)

自然性を図で表すとこんな感じです．

$$\begin{array}{ccc} P(T) & \xrightarrow{t_T} & Q(T) \\ {\scriptstyle P(u)}\downarrow & \circlearrowleft & \downarrow{\scriptstyle Q(u)} \\ P(U) & \xrightarrow[t_U]{} & Q(U) \end{array}$$

この長方形をすんなりイメージできるようになってください．

自然性をこれからどんどん使っていきます！ただし，少し拡張した形で使います．自由型変数や自由変数があったとしても，項の内部においてそれらが束縛されているとしたら，$t : \forall X.\, P(X) \to Q(X)$ とか $u : T \to U$ などと判断していいとします．細かい拡張は皆さんに任せます．

ちなみに，自然性という言葉は圏論に由来しています．いろんな数学の分野においてこの種の性質は見つかるのです．すっかり有名になった**米田の補題** (Yoneda lemma) も圏論に由来していますが，自然性を使えばすぐに証明できます．

補題（米田の補題）． P が共変関手であるとき，任意の型 T について $P(T) \cong \forall X.\, (T \to X) \to P(X)$ である．

証明.

$$\begin{aligned}
&\mathrm{fw} : P(T) \to (\forall X.\ (T \to X) \to P(X)) \\
&\mathrm{fw} \triangleq \lambda\, x : P(T).\ \Lambda X.\ \lambda\, y : T \to X.\ P(y)\ x \\
&\mathrm{bw} : (\forall X.\ (T \to X) \to P(X)) \to P(T) \\
&\mathrm{bw} \triangleq \lambda\, z : (\forall X.\ (T \to X) \to P(X)).\ z_T\ \mathrm{id}_T
\end{aligned}$$

とします．このとき，

$$\begin{aligned}
\mathrm{bw} \circ \mathrm{fw} &\equiv \lambda\, x : P(T).\ (\lambda\, y : T \to T.\ P(y)\ x)\ \mathrm{id}_T \\
&\equiv \mathrm{id}_{P(T)} \qquad \because \text{(SystemF-Functor-Id)}
\end{aligned}$$

であり，かつ

$$\begin{aligned}
\mathrm{fw} \circ \mathrm{bw} &\equiv \lambda\, z : (\forall X.\ (T \to X) \to P(X)).\ \Lambda X.\ \lambda\, y : T \to X.\ P(y)\ (z_T\ \mathrm{id}_T) \\
&\equiv \lambda\, z : (\forall X.\ (T \to X) \to P(X)).\ \Lambda X.\ \lambda\, y : T \to X.\ z_X\ (y \circ \mathrm{id}_T) \quad (\because \textbf{自然性}) \\
&\equiv \mathrm{id}_{\forall X.\ (T \to X) \to P(X)}
\end{aligned}$$

であるので，$P(T) \cong \forall X.\ (T \to X) \to P(X)$ です． (Q.E.D.)

簡単に証明できましたね．

7 データ型をつくろう

Haskell や Caml などの一般的なプログラミング言語は，みな関数型や全称量化型以外にもいわゆる代数的データ型を持っています．ここで使う System F にもすこし手間をかければ簡単にそのようなデータ型を搭載できます．でもそんなデータ型は本当に必要なのでしょうか？実は，関数型と全称量化型があればデータ型として十分な機能を果たせます．ベーム氏とベラルドゥッチ氏は共著論文 [2] において，データ型を関数型と全称量化型だけを用いて表現し，多くの知見をもたらしました．彼らの見つけたデータ型の一般的な表現方法を**ベーム・ベラルドゥッチ・エンコーディング** (Böhm-Berarducci encoding) といいます．これは，型を無視すれば，型無しラムダ計算における **チャーチ・エンコーディング** (Church encoding) に対応しているので，チャーチ・エンコーディングと呼ばれることも多々ありますが，ここではベーム・ベラルドゥッチ・エンコーディングと呼ぶことにしましょう．

さて，**自然性**を使ってさっそく重要な補題を証明しましょう．

補題（型の一般的別表現）. 任意の型 T について、$T \cong \forall X.\ (T \to X) \to X$ である．

証明.

$$\begin{aligned}
&\mathrm{fw} : T \to (\forall X.\ (T \to X) \to X) \\
&\mathrm{fw} \triangleq \lambda\, x : T.\ \Lambda X.\ \lambda\, y : T \to X.\ y\ x \\
&\mathrm{bw} : (\forall X.\ (T \to X) \to X) \to T \\
&\mathrm{bw} \triangleq \lambda\, z : (\forall X.\ (T \to X) \to X).\ z_T\ \mathrm{id}_T
\end{aligned}$$

とします．このとき，

$$\begin{aligned}
\mathrm{bw} \circ \mathrm{fw} &\equiv \lambda\, x : T.\ (\Lambda X.\ \lambda\, y : T \to X.\ y\ x)_T\ \mathrm{id}_T \\
&\equiv \mathrm{id}_A
\end{aligned}$$

であり，かつ

$$\begin{aligned}\mathrm{fw} \circ \mathrm{bw} &\equiv \lambda\, z : (\forall X.\, (T \to X) \to X).\, \Lambda X.\, \lambda\, y : T \to X.\, (y \circ z_T)\, \mathrm{id}_T) \\ &\equiv \lambda\, z : (\forall X.\, (T \to X) \to X).\, \Lambda X.\, \lambda\, y : T \to X.\, (z_X \circ (y\circ))\, \mathrm{id}_T) \quad (\because \textbf{自然性}) \\ &\equiv \lambda\, z : (\forall X.\, (T \to X) \to X).\, \Lambda X.\, \lambda\, y : T \to X.\, z_X\ (y \circ \mathrm{id}_T) \\ &\equiv \mathrm{id}_{\forall X.\ (T\to X)\to X}\end{aligned}$$

であるので，$T \cong \forall X.\, (T \to X) \to X$ です．なお，自然性を使った部分について補足しておくと，$P(X) = T \to X$ としたときに，$t : X \to X'$ について $P(t) = \lambda\, x : T \to X.\, t \circ x$ であるということを利用しています． (Q.E.D.)

この補題はとても強力です．この補題を用いて，さまざまなデータ型を関数型と全称量化型のみによって表現していきましょう．

まずは直積です．とりあえず，直積を System F 上できちんと（型推論の規則を追加したり，ペアのための演繹規則を追加したりして）定義したつもりで議論してみましょう．このとき，

$$T \times U \cong \forall X.\, (T \times U \to X) \to X$$

となります．そして（直積をまともに定義していれば）

$$T \times U \to X \cong T \to U \to X$$

が成り立っています．ということは，

$$T \times U \cong \forall X.\, (T \to U \to X) \to X$$

が成り立っているのです．ここで右辺を見るとめでたいことに関数型と全称量化型しかありません！ということで，直積をわざわざ別個に定義しなくても，

$$T \times U \triangleq \forall X.\, (T \to U \to X) \to X$$

というふうにして直積がつくれてしまうのです！

次は直和です．これもさっきのまったく同様の議論をすればいいのです．直積と直和をきちんと定義したつもりになると，

$$\begin{gathered}T + U \cong \forall X.\, (T + U \to X) \to X \\ T + U \to X \cong T \to X \times U \to X \\ (T \to X \times U \to X) \to X \cong (T \to X) \to (U \to X) \to X \\ \therefore T + U \cong \forall X.\, (T \to X) \to (U \to X) \to X\end{gathered}$$

またしてもめでたく右辺が関数型と全称量化型だけになりました！ということで，直和をわざわざ別個に定義しなくても，

$$T + U \triangleq \forall X.\, (T \to X) \to (U \to X) \to X$$

というふうにして直和がつくれてしまうのです！

型をつくったので，直積と直和に対する処理もきちんとつくっておきましょう．

$$
\begin{aligned}
\mathrm{pair} &: \forall Y.\, \forall Z.\, Y \to Z \to (Y \times Z) \\
\mathrm{pair} &\triangleq \Lambda Y.\, \Lambda Z.\, \lambda\, y : Y.\, \lambda\, z : Z.\, \Lambda X.\, \lambda\, x : Y \to Z \to X.\, x\; y\; z \\
\mathrm{first} &: \forall Y.\, \forall Z.\, (Y \times Z) \to Y \\
\mathrm{first} &\triangleq \Lambda Y.\, \Lambda Z.\, \lambda\, x : Y \times Z.\, x_Y\; (\lambda\, y : Y.\, \lambda\, z : Z.\, y) \\
\mathrm{second} &: \forall Y.\, \forall Z.\, (Y \times Z) \to Y \\
\mathrm{second} &\triangleq \Lambda Y.\, \Lambda Z.\, \lambda\, x : Y \times Z.\, x_Z\; (\lambda\, y : Y.\, \lambda\, z : Z.\, z) \\
\mathrm{left} &: \forall Y.\, \forall Z.\, Y \to Y + Z \\
\mathrm{left} &\triangleq \Lambda Y.\, \Lambda Z.\, \lambda\, y : Y.\, \Lambda X.\, \lambda\, x : Y \to X.\, \lambda\, x' : Z \to X.\, x\; y \\
\mathrm{right} &: \forall Y.\, \forall Z.\, Z \to Y + Z \\
\mathrm{right} &\triangleq \Lambda Y.\, \Lambda Z.\, \lambda\, z : Z.\, \Lambda X.\, \lambda\, x : Y \to X.\, \lambda\, x' : Z \to X.\, x'\; z \\
\mathrm{either} &: \forall Y.\, \forall Z.\, \forall X.\, (Y + Z) \to (Y \to X) \to (Z \to X) \to X \\
\mathrm{either} &\triangleq \Lambda Y.\, \Lambda Z.\, \Lambda X.\, \lambda\, x : Y + Z.\, x_X
\end{aligned}
$$

いろいろ確かめてみてくださいね．

ここでちょっと遠くから眺めてみましょう．Haskell において，直積型 $T \times U = (T, U)$ のデータコンストラクタは

$$(,) :: T \to U \to (T, U)$$

です．また，直和型 $T + U =$ Either T U のデータコンストラクタは

$$
\begin{aligned}
&\mathsf{Left} :: T \to \mathsf{Either}\ T\ U \\
&\mathsf{Right} :: U \to \mathsf{Either}\ T\ U
\end{aligned}
$$

です．ここで改めて定義を見直してみましょう．

$$
\begin{aligned}
T \times U &\triangleq \forall X.\, (T \to U \to X) \to X \\
T + U &\triangleq \forall X.\, (T \to X) \to (U \to X) \to X
\end{aligned}
$$

すると，右辺の X をそれぞれ $T \times U$ や $T + U$ に変えてみるとなんだか**データコンストラクタの型と似ています**．まだピンと来なければ再帰的でない代数的データ型をつくっていろいろ考えてみればいいと思います．

さらに別の見方もできます．(T, U) で**パターンマッチング**をして何かの結果を得るのを考えると，$T \to U \to X$ という型の関数を通じて X という型の値が得られます．Either T U で**パターンマッチング**をして何かの結果を得るのを考えると，$T \to X$ という型の関数と $U \to X$ という型の関数を通じて X という型の値が得られます．ここで，またしても定義

$$
\begin{aligned}
T \times U &\triangleq \forall X.\, (T \to U \to X) \to X \\
T + U &\triangleq \forall X.\, (T \to X) \to (U \to X) \to X
\end{aligned}
$$

を見直してみると，パターンマッチングの過程を関数型と全称量化型によって再現しているかのようです．

さて，直積と直和が定義できましたから，ユニット型とボトム型も定義しておきましょう．これもデータコンストラクタを考えれば簡単です．

ユニット型 Unit ＝ () のデータコンストラクタは () :: () ですから，

$$\mathrm{Unit} \triangleq \forall X.\, X \to X$$

と定義できます．そして，Unit の型の項を考えると，どう考えても恒等関数一つしかありません．つまり，

$$\begin{aligned} &\mathrm{unit} : \mathrm{Unit} \\ &\mathrm{unit} \triangleq \Lambda X.\, \lambda\, x : X.\, x \end{aligned}$$

と定義できます．

一方，ボトム型のデータコンストラクタはありませんから，

$$\mathrm{Bottom} \triangleq \forall X.\, X$$

と定義できます．実際 $\mathrm{Bottom} = \forall X.\, X$ は任意の型がどこからともなく得られる「魔法の型」なので，そんな型を持つ項は作れません．ですから，ボトム型の条件を満たしていますね．なお，全称量化型は結構強いので，項をつくれないような型がいろいろあります．たとえば，$\forall X.\, (X \to X) \to X$ や $\forall X.\, X + X$ などです．

なお，今後のために Maybe と各種演算も定義しておきましょう．

$$\begin{aligned} \mathrm{Maybe}\ T &\triangleq T + \mathrm{Unit} \\ \mathrm{just} &: \forall Y.\, Y \to \mathrm{Maybe}\ Y \\ \mathrm{just} &\triangleq \Lambda Y.\, \mathrm{left}_{Y,\mathrm{Unit}} \\ \mathrm{nothing} &: \forall Y.\, \mathrm{Maybe}\ Y \\ \mathrm{nothing} &\triangleq \Lambda Y.\, \mathrm{right}_{Y,\mathrm{Unit}}\ \mathrm{unit} \\ \mathrm{maybe} &: \forall Y.\, \forall X.\, \mathrm{Maybe}\ Y \to X \to (Y \to X) \to X \\ \mathrm{maybe} &\triangleq \Lambda Y.\, \Lambda X.\, \lambda\, z : \mathrm{Maybe}\ Y.\, z_X \end{aligned}$$

再帰的データ型についてはどうでしょうか．本当に関数型と全称量化型でつくれるのでしょうか．

まずは，Haskell 上で，さまざまな再帰的データ型の定義を振り返ってみましょう．

```
data Nat      = Zero | Succ Nat
data [α]      = [] | α : [α]
data Tree α   = Leaf | Node (Tree α) α (Tree α)
data Rose α   = Branch α [Rose α]
```

それぞれのデータコンストラクタの型を考えてみましょう．

```
Zero   :: Nat
Succ   :: Nat → Nat
[]     :: [α]
(:)    :: α → [α] → [α]
Leaf   :: Tree α
```

$$\begin{aligned}\mathsf{Node} &:: \mathsf{Tree}\,\alpha \to \alpha \to \mathsf{Tree}\,\alpha \to \mathsf{Tree}\,\alpha\\ \mathsf{Branch} &:: \alpha \to [\mathsf{Rose}\,\alpha] \to \mathsf{Rose}\,\alpha\end{aligned}$$

さて，直積型や直和型のときはデータコンストラクタの型をもとにして，型を表現できました．今度もその手法が使えないでしょうか．実は使えるのです．

$$\begin{aligned}\mathrm{Nat} &\triangleq \forall X.\ X \to (X \to X) \to X\\ [T] &\triangleq \forall X.\ X \to (T \to X \to X) \to X\\ \mathrm{Tree}\,T &\triangleq \forall X.\ X \to (X \to T \to X \to X) \to X\\ \mathrm{Rose}\,T &\triangleq \forall X.\ (T \to [X] \to X) \to X\end{aligned}$$

自然数型もようやく定義されましたね．

では，各種演算を考えてみましょう．とはいっても全部考えるのはしんどいですから，とりあえずリストについての演算だけつくってみましょう．

$$\begin{aligned}&\mathrm{foldr} : \forall Y.\ \forall X.\ [Y] \to X \to (Y \to X \to X) \to X\\ &\mathrm{foldr} \triangleq \Lambda Y.\ \Lambda X.\ \lambda z : [Y].\ z_X\\ &\quad \mathrm{nil} : \forall Y.\ [Y]\\ &\quad \mathrm{nil} \triangleq \Lambda Y.\ \Lambda X.\ \lambda x : X.\ \lambda x' : Y \to X \to X.\ x\\ &\mathrm{cons} : \forall Y.\ Y \to [Y] \to [Y]\\ &\mathrm{cons} \triangleq \Lambda Y.\ \lambda y : Y.\ \lambda z : [Y].\ \Lambda X.\ \lambda x : X.\ \lambda x' : Y \to X \to X.\ x'\ y\ (z_X\ x\ x')\\ &\mathrm{head} : \forall Y.\ [Y] \to \mathrm{Maybe}\ Y\\ &\mathrm{head} \triangleq \Lambda Y.\ \lambda z : [Y].\ z_{\mathrm{Maybe}\ Y}\ \mathrm{nothing}_Y\ (\lambda y : Y.\ \lambda x : \mathrm{Maybe}\ Y.\ \mathrm{just}_Y\ y)\\ &\quad \mathrm{tail} : \forall Y.\ [Y] \to \mathrm{Maybe}\ [Y]\\ &\quad \mathrm{tail} \triangleq \Lambda Y.\ \lambda z : [Y].\ z_{\mathrm{Maybe}\ [Y]}\ \mathrm{nothing}_{[Y]}\\ &\qquad\qquad (\lambda y : Y.\ \lambda x : \mathrm{Maybe}\ [Y].\ \mathrm{just}_{[Y]}\ (\mathrm{maybe}_{[Y]}\ x\ (\mathrm{cons}_Y\ y\ \mathrm{nil}_Y)\ (\mathrm{cons}_Y\ y)))\end{aligned}$$

foldr の定義を見ると，$[T] = \forall X.\ X \to (T \to X \to X) \to X$ の型がまさしく**折り畳みを表現している**ということに気づきます！ Nat や Tree や Rose の型を見ても，まさしく折り畳みを表現しています！面白いですね．

他の再帰的データ型についての演算も頑張ればつくれると思います．とはいえ，なにか一般的な形はないものでしょうか．実はあります．

$$\begin{aligned}&\mu X.\ P(X) \triangleq \forall X.\ (P(X) \to X) \to X\\ &\quad \mathrm{fold} : \forall X.\ (P(X) \to X) \to \mu X.\ P(X) \to X\\ &\quad \mathrm{fold} \triangleq \Lambda X.\ \lambda y : P(X) \to X.\ \lambda z : \mu X.\ P(X).\ z_X\ y\\ &\qquad \mathrm{in} : P(\mu X.\ P(X)) \to \mu X.\ P(X)\\ &\qquad \mathrm{in} \triangleq \lambda x : P(\mu X.\ P(X)).\ \Lambda X.\ \lambda y : P(X) \to X.\ y\ (P(\mathrm{fold}_X\ y)\ x)\\ &\quad \mathrm{unin} : \mu X.\ P(X) \to P(\mu X.\ P(X))\\ &\quad \mathrm{unin} \triangleq \mathrm{fold}_{P(\mu X.\ P(X))}\ P(\mathrm{in})\end{aligned}$$

詳しくはワドラー氏のテキストファイル [13] などを参考にしてください．

なお，再帰的データ型をこのように表現することは，効率の面でも実益があります．Haskell 上で考えましょう．

```
newtype BList α = BList (∀β. β → (α → β → β) → β)
single  :: α → BList α
single x = BList (λz f → f x z)

(+++)                 :: BList α → BList α → BList α
BList xs +++ BList ys = BList (λz f → xs (ys z f) f)

toList           :: BList α → [α]
toList (BList xs) = xs [] (:)
```

このようにしたとき，次の計算の効率を比較してみましょう．

```
goA, goB :: Int → BList Int
goA 1 = single 1
goA n = single n +++ goA (n − 1)
goB 1 = single 1
goB n = goB (n − 1) +++ single n

goC, goD :: Int → [Int]
goC 1 = [1]
goC n = [n] ++ goC (n − 1)
goD 1 = [1]
goD n = goD (n − 1) ++ [n]

manyA, manyB, manyC, manyD :: [Int]
manyA = toList (goA 100000)
manyB = toList (goB 100000)
manyC = goC 100000
manyD = goD 100000
```

実験してみると，manyA, manyB, manyC が計算があっという間に終わるのに対して，manyD は計算が恐ろしく長くかかるでしょう．[20] なぜでしょうか．実は $+\!+\!+$ と $+\!+$ には大きな違いがあるのです．$xs +\!+\!+ ys$ の計算量は常に定数時間ですが，$xs +\!+ ys$ の計算量は xs の長さに比例します．それゆえに，manyD だけ恐ろしく時間がかかったのです．

このようなことをするためのテクニックとして，**差分リスト** (difference list) を知っている人も多いと思います．少し考えれば，BList が差分リストの一般化となっていることが分かります．差分リスト自体はリスト専用の技法ですが，ベーム・ベラルドゥッチ・エンコーディングはどんな再帰的データ型にも使える技法ですから，二分木や多分木やもっと複雑なデータ型でも，差分リストのような計算の効率化ができるのです．素晴らしいですね．

関数型と全称量化型を使って**存在型** (existential type) も表現できます．Haskell でも ExistentialQuantification という GHC 拡張を使えば存在型を使えます．そういえば Haskell でも存在型の宣言には全称量化型が使われています．どういうことでしょうか．実は，存在型と全称量化型には密接なかかわりがあるのです．

[20] 実験するときは，print (length manyA) などとするといいと思います．

たとえば，学校で抜き打ちテストが行われることを考えましょう．抜き打ちテストがいつの日にか行われるということだけが分かっているとします．このとき，先生は**任意**の日にテストを実施できます．一方，生徒にとってはいつだか分からない**ある**日にテストが実施されます．与える側にとっては全称量化的でも，与えられる側にとっては存在量化的であるのです．

このことを型理論上で表現すると，次のようになります．

$$\forall X.\, P(X) \to T \cong (\exists X.\, P(X)) \to T$$

関数の外から見ると任意の X を関数に渡せて，関数の中から見るとある X が引数に来るという具合です．

これで Haskell の存在型も説明できます．次のような単純な存在型を考えましょう．

$$\textbf{data}\ \mathsf{Any}\ =\ \forall\,\alpha.\,\mathsf{Any}\,\alpha$$

ここで，データコンストラクタの型を考えると，

$$\mathsf{Any}\ ::\ \forall\,\alpha.\,\alpha\ \to\ \mathsf{Any}$$

となっています．ここで，先ほどの同型関係を用いると，

$$\forall\,\alpha.\,\alpha\ \to\ \mathsf{Any}\ \cong\ (\exists\,\alpha.\,\alpha)\ \to\ \mathsf{Any}$$

となるのです（実際には∃という構文は Haskell にはありませんが）．

存在型とは一体どんなものなのでしょうか．存在型を扱う簡単な方法としては，$\exists X.\, P(X)$ を，型 T と $P(T)$ 型の値の組とみなす方法があります．あまり文献が多くありませんが，[13] に，System F に存在型を追加して扱う方法が書いてあります．

では存在型を System F 上につくったつもりで考えましょう．**型の一般的別表現**と先ほどの同型関係を使うと，

$$\begin{aligned}\exists X.\, T &\cong \forall Y.\, ((\exists X.\, T) \to Y) \to Y \\ &\cong \forall Y.\, (\forall X.\, T \to Y) \to Y\end{aligned}$$

よって，存在型をわざわざ作らなくても

$$\exists X.\, T \triangleq \forall Y.\, (\forall X.\, T \to Y) \to Y$$

と定義すればいいことになります．存在型についての演算も定義しておきましょう．[21]

$$\begin{aligned}&\mathrm{makeExists}_P : P(T) \to \exists X.\, P(X) \\ &\mathrm{makeExists}_P \triangleq \lambda\, x : P(T).\, \Lambda Y.\, \lambda\, y : (\forall X.\, P(X) \to Y).\, y_T\ x \\ &\mathrm{withExists}_P : \forall Y.\, (\exists X.\, P(X)) \to (\forall X.\, P(X) \to Y) \to Y \\ &\mathrm{withExists}_P \triangleq \Lambda Y.\, \lambda\, x : (\exists X.\, P(X)).\, x_Y\end{aligned}$$

関数型と全称量化型だけで本当に多くのデータ型が表現できることが分かりました．System F はガチでヤバいですね！

[21] いまつくっている System F では，型を受け取る型について全称量化をすることができないので，以下のようにしておきます．

8 もっと再帰しよう

ここまでつくってきた System F は，再帰を自由に書ける機能がないので，計算が必ず有限時間で終了するということが保証されていました．そのおかげで，プログラムについてのさまざまな推論が楽になっていました．

しかし，それと引き換えに，計算能力が大きく減ってしまいました．つまり，**チューリング完全** (Turing complete) で無くなってしまったのです．再帰を自由に書ける機能を追加すれば，チューリング完全になります．そもそも再帰すらロクにできないプログラミング言語なんてイヤですね！そこで，System F に次の関数を追加すればいいのです．

$$\mathrm{fix} : \forall X.\ (X \to X) \to X$$

この関数は普通に Haskell でプログラミングをしていても触れるでしょうから，分かると思います．

構文論と意味論，そして二項関係的解釈のそれぞれにきちんと修正を加えなければなりません．

まずは構文論を修正しましょう．項に fix を追加して

$$(\text{項}) ::= (\text{変数}) \mid \lambda\,(\text{変数}) : (\text{型}).\ (\text{項}) \mid (\text{項})\ (\text{項}) \mid \Lambda(\text{型変数}).\ (\text{項}) \mid (\text{項})_{(\text{型})} \mid \mathrm{fix}$$

とします．そして型推論のルールに次を追加します．

$$\frac{}{\vec{X}; \vec{x} \vdash \mathrm{fix} : \forall X.\ (X \to X) \to X} \qquad \text{(Fix-Trivial)}$$

計算のルールも増やしましょう．

$$\mathrm{fix}_T\ f \Rightarrow_\beta f\ (\mathrm{fix}_T\ f)$$

計算の順序を間違えると容易に無限ループになりますが，そこは Haskell と同様に上手くやればいいだけです．

構文論の修正は簡単でしたね．

次は意味論を修正しましょう．これについてはそれなりに大きい変更が必要です．

まずは値の集合 $\mathbf{D}_A$ に対していくつかの**要請**を課していきます．

各 $A \in \mathbf{U}$ について，$\mathbf{D}_A$ が半順序 $\sqsubseteq_A$ により順序付けられていることを要請します．この半順序を**近似順序** (approximation order) といいます．なお，$\sqsubseteq_A$ が**半順序** (partially ordered set, poset) であるとは，

反射律 任意の $a \in \mathbf{D}_A$ について，$a \sqsubseteq_A a$
反対称律 任意の $a, b \in \mathbf{D}_A$ について，$a \sqsubseteq_A b$ かつ $b \sqsubseteq_A a$ ならば $a = b$
推移律 任意の $a, b, c \in \mathbf{D}_A$ について，$a \sqsubseteq_A b$ かつ $b \sqsubseteq_A c$ ならば $a \sqsubseteq_A c$

を満たしているということです．$a \sqsubseteq_A a'$ かつ $a \neq a'$ であるとき $a \sqsubset_A a'$ と書きます．近似順序についての条件はこれから付け足していきますが，$a \sqsubset_A b$ は「a より b のほうがより計算が進んでいる」あるいは「a より b のほうが情報量が多い」ということを表していると思ってください．

$\mathbf{D}_A$ が $\sqsubseteq_A$ についての最小元 $\bot_A$ を持つことも要請します．この元を A の**ボトム** (bottom) といいます．

単調増加の無限数列のことを**ω鎖**（オメガさ） (omega chain) といいます．任意の型値 $A \in \mathbf{U}$ と，$\mathbf{D}_A$ の元からなる任意の ω 鎖 $a_1 \sqsubseteq_A a_2 \sqsubseteq_A a_3 \sqsubseteq_A \cdots$ について，その**上限** (supremum) $\bigsqcup_i a_i$ が存在することを要請します．ただし，$\{a_i\}$ の上限 $\bigsqcup_i a_i$ とは，

任意の i について $a_i \sqsubseteq_A s$ であり，任意の $s' \sqsubset_A s$ について $s' \sqsubset_A a_k$ となる k が存在する

を満たす $s \in \mathbf{D}_A$ のことです．

更に次のことを要請します．

単調性 $a \sqsubseteq_A a'$ ならば $\phi_{A,B}\, c\, a \sqsubseteq_B \phi_{A,B}\, c\, a'$ (Func-Monotonous)

連続性 任意の ω 鎖 $\{a_i\}$ について $\phi_{A,B}\, c\, (\bigsqcup_i a_i) = \bigsqcup_i (\phi_{A,B}\, c\, a_i)$ (Func-Continuous)

なお，ω 鎖 $\{a_i\}$ について (Func-Monotonous) より $\phi_{A,B}\, c\, a_1 \sqsubseteq_B \phi_{A,B}\, c\, a_2 \sqsubseteq_B \phi_{A,B}\, c\, a_3 \sqsubseteq_B \cdots$ が成り立つので $\{\phi_{A,B}\, c\, a_i\}$ も ω 鎖になっている，ということに注意してください．

また，関数型と全称量化型の近似順序については以下のことを要請します．

$$c \sqsubseteq_{A\to B} c' \iff \text{「任意の } a \in \mathbf{D}_A \text{ について } \phi_{A,B}\, c\, a \sqsubseteq_B \phi_{A,B}\, c'\, a\text{」}$$
$$d \sqsubseteq_{\forall F} d' \iff \text{「任意の } A \in \mathbf{U} \text{ について } \Phi_{\forall F}\, d\, A \sqsubseteq_{F(A)} \Phi_{\forall F}\, d'\, A\text{」}$$

ここから次のことも容易に導かれます．

$$\text{任意の } a \in \mathbf{D}_A \text{ について } \phi_{A,B} \perp_{A\to B} a = \perp_B$$
$$\text{任意の } A \in \mathbf{U} \text{ について } \Phi_F \perp_{\forall F} A = \perp_{F(A)}$$

以上をまとめると，値の集合について，近似順序があり，その最小元があり，ω 鎖が上限をもち，関数適用が単調性と連続性という綺麗な性質をもち，関数型と全称量化型の近似順序が定まっている，ということを要請しました．ちなみに，「半順序があり，その最小元があり，ω 鎖が上限をもつ」ような集合を，計算機科学では一般に**領域** (domain) といいます．そして，（特定の条件を満たす）領域について研究する分野を**領域理論** (domain theory) といいます．

次は fix に対する解釈を追加しましょう．次のように定義すればいいです．

$$[\![\mathrm{fix}]\!]\, \vec{A}\, \vec{a} \triangleq \Psi_F\, (B \mapsto \psi_{B\to B,B}\, (c \mapsto \bigsqcup_i ((\phi_{B,B}\, c)^i \perp_B))) \quad (\text{ただし } F(B) = \mathbf{D}_{(B\to B)\to B})$$

ただし，$(\phi_{B,B}\, c)^i \perp_B = (\phi_{B,B}\, c \circ \phi_{B,B}\, c \circ \cdots \circ \phi_{B,B}\, c) \perp_B$ とします．

ちなみに $\{(\phi_{B,B}\, c)^i \perp_B\}$ が ω 鎖であることは容易に証明できます．

証明． $\perp_B$ は最小元なので $\perp_B \sqsubseteq_B \phi_{B,B}\, c \perp_B$ です．また，$(\phi_{B,B}\, c)^k \perp_B \sqsubseteq_B (\phi_{B,B}\, c)^{k+1} \perp_B$ ならば (Func-Monotonous) より $(\phi_{B,B}\, c)^{k+1} \perp_B \sqsubseteq_B (\phi_{B,B}\, c)^{k+2} \perp_B$ です．よって帰納的に，

$$\perp_B \sqsubseteq_B \phi_{B,B}\, c \perp_B \sqsubseteq_B (\phi_{B,B}\, c \circ \phi_{B,B}\, c) \perp_B \sqsubseteq_B (\phi_{B,B}\, c \circ \phi_{B,B}\, c \circ \phi_{B,B}\, c) \perp_B \sqsubseteq_B \cdots$$

が成立します． (Q.E.D.)

さて，値の集合にいろいろな要請を与えてきたのは，紛れもなく fix を定義するためです．この定義を直感的に理解するには，遅延評価などで計算を保留していた部分がだんだん明らかになっていく，というふうに理解すればいいと思います．

とにかく，fix に満たしてほしい性質はといえばもちろん不動点であることです．これはすぐに証明できます．

証明. $\vec{X}; \vec{x} \vdash u : T \to T$ であり，$\langle \vec{A}, \vec{a} \rangle$ が $\vec{X}, \vec{x}$ にかんする環境対であるとします．このとき，$F(C) = (C \to C) \to C, B = [\![T]\!]\ \vec{A}, c = [\![u]\!]\ \vec{A}\ \vec{a}$ とおくと，

$$\begin{aligned}
[\![\mathrm{fix}_T\ u]\!]\ \vec{A}\ \vec{a} &= \phi_{(B\to B),B}\ (\Phi_F\ \mathrm{fix}\ B)\ c \\
&= \bigsqcup_i ((\phi_{B,B}\ c)^i \bot_B) \\
&= \bigsqcup_i (\phi_{B,B}\ c\ ((\phi_{B,B}\ c)^i \bot_B)) \\
&= \phi_{B,B}\ c\ (\bigsqcup_i ((\phi_{B,B}\ c)^i \bot_B)) \quad \because \text{(Func-Continuous)} \\
&= [\![u\ (\mathrm{fix}_T\ u)]\!]\ \vec{A}\ \vec{a}
\end{aligned}$$

が成立します． (Q.E.D.)

次は二項関係の意味論について考えてみましょう．一番の問題はパラメトリシティがそのまま成立するかどうかです．実は成立しません．単純な反例を紹介します．

> $A = [\![\mathrm{Unit}]\!]\ \varnothing$ とし，$a = [\![\mathrm{unit}]\!]\ \varnothing\ \varnothing$ とします．このとき，$a \in \mathbf{D}_A$ かつ $a \neq \bot_A$ です．そして，$\mathcal{A} : \mathbf{D}_A \leftrightarrow \mathbf{D}_A$ を $\mathcal{A} = \{\langle x, a \rangle \mid x \in \mathbf{D}_A\}$ により定めます．
> パラメトリシティより $\langle [\![\mathrm{fix}]\!]\ \varnothing\ \varnothing, [\![\mathrm{fix}]\!]\ \varnothing\ \varnothing \rangle \in [\![\mathrm{Unit}]\!]\ \varnothing$ であるので，$\langle [\![\mathrm{fix}_{\mathrm{Unit}}]\!]\ \varnothing\ \varnothing, [\![\mathrm{fix}_{\mathrm{Unit}}]\!]\ \varnothing\ \varnothing \rangle \in (\mathcal{A} \to \mathcal{A}) \to \mathcal{A}$ が成立しています．
> ここで，$c, c' : \mathbf{D}_{A\to A}$ が任意の $x \in \mathbf{D}_A$ について $\phi_{A,A}\ c\ x = \bot_A, \phi_{A,A}\ c'\ x = x$ を満たすとします．すると，$\langle x, x' \rangle \in \mathcal{A}$ のとき $\phi_{A,A}\ c\ x' = x' = \bot_A$ なので $\langle \phi_{A,A}\ c\ x, \phi_{A,A}\ c'\ x' \rangle \in \mathcal{A}$ です．
> よって，$\langle \phi_{A\to A,A}\ ([\![\mathrm{fix}_{\mathrm{Unit}}]\!]\ \varnothing\ \varnothing)\ c, \phi_{A\to A,A}\ ([\![\mathrm{fix}_{\mathrm{Unit}}]\!]\ \varnothing\ \varnothing)\ c' \rangle \in \mathcal{A}$ であるはずです．しかし，$\phi_{A\to A,A}\ ([\![\mathrm{fix}_{\mathrm{Unit}}]\!]\ \varnothing\ \varnothing)\ c' = \bot_A \neq a$ であるので，矛盾です．

これは大変です．パラメトリシティの定理をぜひとも修正しなければなりません．その修正というのは，二項関係を**連続かつ正格**なものに限定することです．

まず，二項関係 $\mathcal{A} : \mathbf{D}_A \leftrightarrow \mathbf{D}_{A'}$ が**連続** (continuous) であるとは，$\mathbf{D}_A$ の値からなる任意の ω 鎖 $x_1 \sqsubseteq_A x_2 \sqsubseteq_A \cdots$ と，$\mathbf{D}_{A'}$ の値からなる任意の ω 鎖 $x'_1 \sqsubseteq_{A'} x'_2 \sqsubseteq_{A'} \cdots$ について，任意の i について $\langle x_i, x'_i \rangle \in \mathcal{A}$ が成立しているならば，必ず $\langle \bigsqcup_i x_i, \bigsqcup_i x'_i \rangle \in \mathcal{A}$ が成立するということです．これは，先ほど定義した関数のときの連続性と似ていますね．

次に，$\mathcal{A} : \mathbf{D}_A \leftrightarrow \mathbf{D}_{A'}$ が**正格** (strict) であるとは，$\langle \bot_A, \bot'_A \rangle \in \mathcal{A}$ であるということです．単純ですね．

二項関係を連続かつ正格であるものに限定するということをきちんと整理しておきましょう．

まず，$\forall \mathcal{F}$ を

$$\begin{aligned}
\mathcal{F} \triangleq \{\langle d, d' \rangle \mid &\text{任意の } A, A' \text{と連続かつ正格な二項関係} \mathcal{A} : \mathbf{D}_A \leftrightarrow \mathbf{D}_{A'} \text{ について} \\
&\langle \Phi_F\ d\ A, \Phi_{F'}\ d'\ A' \rangle \in \mathcal{F}(\mathcal{A})\}
\end{aligned}$$

と定義しなおします．ただし，$F : \mathbf{U} \to \mathbf{U}, F' : \mathbf{U} \to \mathbf{U}$ であり，$\mathcal{F}$ が二項関係 $\mathcal{A} : \mathbf{D}_A \leftrightarrow \mathbf{D}_{A'}$ を取って二項関係 $\mathcal{F}(\mathcal{A}) : \mathbf{D}_{F(A)} \leftrightarrow \mathbf{D}_{F'(A')}$ を返す関数であるとします．

そして，$\vec{X}$ に対する関係環境 $\vec{\mathcal{A}}$ は，「任意の $X_i \in \vec{X}$ について $\vec{\mathcal{A}}\ [\![X_i]\!]$ が連続かつ正格である」ということを満たさなければならないということにします．

お疲れ様でした．このようにすると，パラメトリシティがきちんと成立します．証明の細かい変更はみなさんに任せますが，fix についてだけ証明しておきます．

補題（*fix* についてのパラメトリシティ）. $\langle [\![\mathrm{fix}]\!]\,\varnothing\,\varnothing, [\![\mathrm{fix}]\!]\,\varnothing\,\varnothing\rangle \in [\![\forall X.\,(X \to X) \to X]\!]\,\varnothing$ である.

証明. $\mathcal{A} : \mathbf{D}_A \leftrightarrow \mathbf{D}_{A'}$ を連続かつ正格な二項関係とし，$c : \mathbf{D}_{A\to A}$ と $c' : \mathbf{D}_{A'\to A'}$ が $\langle c, c'\rangle \in \mathcal{A} \to \mathcal{A}$ を満たしているとします.

$\mathcal{A}$ は正格なので，$\langle \bot_A, \bot_A\rangle \in \mathcal{A}$ です．さらに $\langle c, c'\rangle \in \mathcal{A} \to \mathcal{A}$ なので，これを繰り返し使うと，任意の i について $\langle (\phi_{A,A}\ c)^i \bot_A, (\phi_{A',A'}\ c')^i \bot_{A'}\rangle \in \mathcal{A}$ が得られます．よって，$\mathcal{A}$ の連続性から，$\langle \bigsqcup_i ((\phi_{A,A}\ c)^i \bot_A), \bigsqcup_i ((\phi_{A',A'}\ c')^i \bot_{A'})\rangle \in \mathcal{A}$ が成立します.

ゆえに，$\langle [\![\mathrm{fix}]\!]\,\varnothing\,\varnothing, [\![\mathrm{fix}]\!]\,\varnothing\,\varnothing\rangle \in [\![\forall X.\,(X \to X) \to X]\!]\,\varnothing$ が成立します. (Q.E.D.)

これで一安心ですが，定義のさまざまな修正により，これまで成立していたいろいろなことが微妙に成立しなくなります．ラムダ式から二項関係をつくる場合，連続性は自動的に満たされているので問題ありませんが，正格性はかなり強い条件で，なかなか満たされません．たとえば，**型の一般的別表現**なども成り立たなくなります．なお，fix だけでなく強制的な正格評価（seq）を導入した場合もさらに修正が必要です（詳しくは [5] を参照してください）.

Haskell にパラメトリシティを応用する際には，正格性に注意しなければなりませんが，それほど深刻な問題にならないことも多いので，落ち着いて考えていけばいいと思います．なお，この記事ではこれから**自然性**を使う場面がありますが，正格性についてはあまり気にしないことにします．厳密な議論はみなさんに任せます.

第四章

アプリカティヴ

1　アプリカティヴの基本

「アプリカティヴ」という型クラスは，Haskell の歴史において関手やモナドよりもずっと後に認知されました．そのせいもあってか，アプリカティヴという型クラスはかなり誤解されていると思います．アプリカティヴは本当に単純であり，単純であるからこそ姿を捉えがたいのです．アプリカティヴの真の姿に少しでも迫っていきましょう．

以下の内容はマクブライド氏とペーターソン氏の共著の論文 [8] を参考にしています．

まず，アプリカティヴの型クラスを定義しましょう．

```
class Functor φ ⇒ Applicative φ where
  pure :: α → φ α
  (⊛) :: φ (α → β) → φ α → φ β    — "ap"
infixl 4 ⊛
```

アプリカティヴ則は次の通りです．

純粋	pure id ⊛ x ≡ x	(Ap-Id)
合成	pure (∘) ⊛ u ⊛ v ⊛ w ≡ u ⊛ (v ⊛ w)	(Ap-Comp)
関数適用	pure f ⊛ pure x ≡ pure (f x)	(Ap-Apply)
交換	u ⊛ pure y ≡ pure ($ y) ⊛ u	(Ap-Swap)

このアプリカティヴ則は，初めて見る人にとっては謎に満ちていると思います．何となく理解できないことはないけれども，この 4 つが選ばれている意味がどうにも解せないという人も多いと思います．その疑問は後で解決されてゆくはずです．

さて，アプリカティヴは関手のサブクラスですね．では，pure と ⊛ から fmap を作れるでしょうか？型を考えると，次のようにすれば良さそうです．

```
fmap      :: Applicative φ ⇒ (α → β) → φ α → φ β
fmap f x = pure f ⊛ x
```

本当にファンクタ則を満たしていることを証明してみましょう．

$$
\begin{aligned}
&\mathsf{fmap}\ \mathsf{id}\ x \equiv \mathsf{pure}\ \mathsf{id} \circledast x \equiv x && \because \text{(Ap-Id)}\\
&\mathsf{fmap}\ (g \circ f)\ x\\
&\equiv \mathsf{pure}\ (g \circ f) \circledast x\\
&\equiv \mathsf{pure}\ (\circ) \circledast \mathsf{pure}\ g \circledast \mathsf{pure}\ f \circledast x && \because \text{(Ap-Apply)}\\
&\equiv \mathsf{pure}\ g \circledast (\mathsf{pure}\ f \circledast x) && \because \text{(Ap-Comp)}\\
&\equiv \mathsf{fmap}\ g\ (\mathsf{fmap}\ f\ x)\\
&\equiv (\mathsf{fmap}\ g \circ \mathsf{fmap}\ f)\ x
\end{aligned}
$$

きちんと満たしていますね．任意のデータ型について，正しい型を持ちファンクタ則をきちんと満たしている fmap は高々一つしかないはずなので，

関手との関係 $\mathsf{fmap}\ f\ x \equiv \mathsf{pure}\ f \circledast x$ (Ap-Fmap)

が導けます．なおこれを踏まえれば，アプリカティヴ則は

$$
\begin{aligned}
&\mathsf{id} \mathbin{\langle\$\rangle} x \equiv x\\
&(\circ) \mathbin{\langle\$\rangle} u \circledast v \circledast w \equiv u \circledast (v \circledast w)\\
&f \mathbin{\langle\$\rangle} \mathsf{pure}\ x \equiv \mathsf{pure}\ (f\ x)\\
&u \circledast \mathsf{pure}\ y \equiv (\$\ y) \mathbin{\langle\$\rangle} u
\end{aligned}
$$

と書き直すこともできます．

さらに考えましょう．fmap の型は $(\alpha \to \beta) \to \varphi\,\alpha \to \varphi\,\beta$ ですが，$(\alpha \to \beta \to \gamma) \to \varphi\,\alpha \to \varphi\,\beta \to \varphi\,\gamma$ や $(\alpha \to \beta \to \gamma \to \delta) \to \varphi\,\alpha \to \varphi\,\beta \to \varphi\,\gamma \to \varphi\,\delta$ などの型の関数を，pure と $\circledast$ を使って作れるでしょうか？次のようにして作れます．

$$
\begin{aligned}
&\mathsf{fmap2} :: \mathsf{Applicative}\ \varphi \Rightarrow (\alpha \to \beta \to \gamma) \to \varphi\,\alpha \to \varphi\,\beta \to \varphi\,\gamma\\
&\mathsf{fmap2}\ f\ x\ y = \mathsf{pure}\ f \circledast x \circledast y\\
&\mathsf{fmap3} :: \mathsf{Applicative}\ \varphi \Rightarrow (\alpha \to \beta \to \gamma \to \delta) \to \varphi\,\alpha \to \varphi\,\beta \to \varphi\,\gamma \to \varphi\,\delta\\
&\mathsf{fmap3}\ f\ x\ y\ z = \mathsf{pure}\ f \circledast x \circledast y \circledast z
\end{aligned}
$$

同様にしていくらでも続けていけます．fmap だけでは fmap2 や fmap3 のような関数は（型だけを考えても）作れません．アプリカティヴは，ある意味，関手を一引数向けから多引数向けへと単純に拡張したものとみなすこともできるのです．

アプリカティヴ則は比較的綺麗ですが，このアプリカティヴ則を使って何かを証明するのはなかなか難しそうに見えます．特に，所々ポイントフリーなのがなかなか取っ付きにくいですね．今後のために，便利な定理を証明しておきましょう．一番目と二番目は，三番目を導くための補題です．

$$
\begin{aligned}
&\mathsf{pure}\ f \circledast (\mathsf{pure}\ g \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_n)\\
&\equiv \quad \mathsf{pure}\ (\lambda\, y_1\ y_2\ \cdots\ y_n \to f\ (g\ y_1\ y_2\ \cdots\ y_n)) \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_n
\end{aligned}
\qquad \text{(Ap-Nest-LemmaA)}
$$

$$
\begin{aligned}
&\mathsf{pure}\ f \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m \circledast (\mathsf{pure}\ g \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_n)\\
&\equiv \quad \mathsf{pure}\ (\lambda\, z_1\ z_2\ \cdots\ z_m\ w_1\ w_2\ \cdots\ w_n \to f\ z_1\ z_2\ \cdots\ z_m\ (g\ w_1\ w_2\ \cdots\ w_n))\\
&\qquad \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_n
\end{aligned}
\qquad \text{(Ap-Nest-LemmaB)}
$$

$$
\begin{aligned}
&\mathsf{pure}\, f \circledast (\mathsf{pure}\, g_1 \circledast x_{1,1} \circledast x_{1,2} \circledast \cdots \circledast x_{1,n_1}) \\
&\qquad \circledast (\mathsf{pure}\, g_2 \circledast x_{2,1} \circledast x_{2,2} \circledast \cdots \circledast x_{2,n_2}) \\
&\qquad \circledast \cdots \\
&\qquad \circledast (\mathsf{pure}\, g_m \circledast x_{m,1} \circledast x_{m,2} \circledast \cdots \circledast x_{m,n_m}) \\
&\equiv \quad \mathsf{pure}\, (\lambda\, y_{1,1}\ y_{1,2}\ \cdots\ y_{1,n_1} \\
&\qquad\quad y_{2,1}\ y_{2,2}\ \cdots\ y_{2,n_2} \\
&\qquad\quad \cdots \\
&\qquad\quad y_{m,1}\ y_{m,2}\ \cdots\ y_{m,n_m}\ \to \\
&\qquad\quad f\ (g_1\ y_{1,1}\ y_{1,2}\ \cdots\ y_{1,n_1}) \\
&\qquad\quad (g_2\ y_{2,1}\ y_{2,2}\ \cdots\ y_{2,n_2}) \\
&\qquad\quad \cdots \\
&\qquad\quad (g_m\ y_{m,1}\ y_{m,2}\ \cdots\ y_{m,n_m})) \qquad \text{(Ap-Nest)}
\end{aligned}
$$

一般性が高いのでちょっと怖そうに見えるかもしれませんが，本当に単純なので落ち着いて見てください．

まずは (Ap-Nest-LemmaA) を証明しましょう．ここでは (Ap-Comp), (Ap-Apply) しか使いません．

$$
\begin{aligned}
&\mathsf{pure}\, f \circledast (\mathsf{pure}\, g \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_n) \\
&\equiv \mathsf{pure}\, (f \circ) \circledast (\mathsf{pure}\, g \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_{n-1}) \circledast x_n && \because \text{(Ap-Comp), (Ap-Apply)} \\
&\equiv \cdots \\
&\equiv \mathsf{pure}\, (((\cdots((f \circ) \circ) \cdots) \circ) \circ) \circledast g \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_n && \because \text{(Ap-Comp), (Ap-Apply)} \\
&\equiv \mathsf{pure}\, ((\cdots((f \circ) \circ) \cdots) \circ) \circ\ g \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_n && \because \text{(Ap-Apply)} \\
&\equiv \mathsf{pure}\, (\lambda\, y_1\ \to\ (\cdots((f \circ) \circ) \cdots) \circ\ g\ y_1) \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_n \\
&\equiv \cdots \\
&\equiv \mathsf{pure}\, (\lambda\, y_1\ y_2\ \cdots\ y_n\ \to\ f\ (g\ y_1\ y_2\ \cdots\ y_n)) \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_n
\end{aligned}
$$

次に (Ap-Nest-LemmaB) を証明しましょう．ここでは (Ap-Comp), (Ap-Apply) に加えて (Ap-Swap)，そして先ほど証明した (Ap-Nest-LemmaA) を使います．

$$
\begin{aligned}
&\mathsf{pure}\, f \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m \circledast (\mathsf{pure}\, g \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_n) \\
&\equiv \mathsf{pure}\, (\circ) \circledast (\mathsf{pure}\, f \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m) \\
&\qquad \circledast (\mathsf{pure}\, g \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_{n-1}) \circledast y_n && \because \text{(Ap-Comp)} \\
&\equiv \cdots \\
&\equiv \mathsf{pure}\, (\circ) \circledast (\mathsf{pure}\, (\circ) \circledast (\cdots (\mathsf{pure}\, (\circ) \\
&\qquad \circledast (\mathsf{pure}\, f \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m)) \cdots)) \\
&\qquad \circledast \mathsf{pure}\, g \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_n && \because \text{(Ap-Comp)} \\
&\equiv \mathsf{pure}\, (\$\ g) \\
&\qquad \circledast (\mathsf{pure}\, (\circ) \circledast (\mathsf{pure}\, (\circ) \circledast (\cdots (\mathsf{pure}\, (\circ) \\
&\qquad \circledast (\mathsf{pure}\, f \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m)) \cdots))) \\
&\qquad \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_n && \because \text{(Ap-Swap)} \\
&\equiv \mathsf{pure}\, ((\$\ g) \circ (\circ)) \\
&\qquad \circledast (\mathsf{pure}\, (\circ) \circledast (\cdots (\mathsf{pure}\, (\circ) \circledast (\mathsf{pure}\, f \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m)) \cdots)) \\
&\qquad \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_n && \because \text{(Ap-Comp), (Ap-Apply)}
\end{aligned}
$$

$$
\begin{aligned}
&\equiv \cdots \\
&\equiv \mathsf{pure}\,((\$\ g) \circ (\circ) \circ (\circ) \circ \cdots \circ (\circ) \circ (\circ)) \\
&\quad \circledast (\mathsf{pure}\, f \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m) \\
&\quad \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_n && \because \text{(Ap-Comp), (Ap-Apply)} \\
&\equiv \mathsf{pure}\,(\lambda\, z_1\ z_2\ \cdots\ z_m\ \to \\
&\quad ((\$\ g) \circ (\circ) \circ (\circ) \circ \cdots \circ (\circ) \circ (\circ))\,(f\ z_1\ z_2\ \cdots\ z_n)) \\
&\quad \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_n && \because \text{(Ap-Nest-LemmaA)} \\
&\equiv \mathsf{pure}\,(\lambda\, z_1\ z_2\ \cdots\ z_m\ \to\ ((\cdots((f\ z_1\ z_2\ \cdots\ z_n \circ)\circ)\cdots)\circ) \circ\ g) \\
&\quad \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_n \\
&\equiv \mathsf{pure}\,(\lambda\, z_1\ z_2\ \cdots\ z_m\ w_1\ \to\ (\cdots((f\ z_1\ z_2\ \cdots\ z_n \circ)\circ)\cdots) \circ\ g\ w_1) \\
&\quad \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_n \\
&\equiv \cdots \\
&\equiv \mathsf{pure}\,(\lambda\, z_1\ z_2\ \cdots\ z_m\ w_1\ w_2\ \cdots\ w_n\ \to\ f\ z_1\ z_2\ \cdots\ z_n\ (g\ w_1\ w_2\ \cdots\ w_n)) \\
&\quad \circledast x_1 \circledast x_2 \circledast \cdots \circledast x_m \circledast y_1 \circledast y_2 \circledast \cdots \circledast y_n
\end{aligned}
$$

最後にいよいよ (Ap-Nest-Lemma) を証明しましょう．今度は (Ap-Nest-LemmaA) と (Ap-Nest-LemmaB) のみを使います．

$$
\begin{aligned}
&\mathsf{pure}\, f \circledast (\mathsf{pure}\, g_1 \circledast x_{1,1} \circledast x_{1,2} \circledast \cdots \circledast x_{1,n_1}) \\
&\qquad \circledast (\mathsf{pure}\, g_2 \circledast x_{2,1} \circledast x_{2,2} \circledast \cdots \circledast x_{2,n_2}) \\
&\qquad \circledast \cdots \\
&\qquad \circledast (\mathsf{pure}\, g_m \circledast x_{m,1} \circledast x_{m,2} \circledast \cdots \circledast x_{m,n_m}) \\
&\equiv \quad \mathsf{pure}\,(\lambda\, y_{1,1}\ y_{1,2}\ \cdots\ y_{1,n_1}\ \to\ f\ (g_1\ y_{1,1}\ y_{1,2}\ \cdots\ y_{1,n_1})) \\
&\qquad \circledast x_{1,1} \circledast x_{1,2} \circledast \cdots \circledast x_{1,n_1} \\
&\qquad \circledast (\mathsf{pure}\, g_2 \circledast x_{2,1} \circledast x_{2,2} \circledast \cdots \circledast x_{2,n_2}) \\
&\qquad \circledast \cdots \\
&\qquad \circledast (\mathsf{pure}\, g_m \circledast x_{m,1} \circledast x_{m,2} \circledast \cdots \circledast x_{m,n_m}) && \because \text{(Ap-Nest-LemmaA)} \\
&\equiv \quad \mathsf{pure}\,(\lambda\, y_{1,1}\ y_{1,2}\ \cdots\ y_{1,n_1}\ y_{2,1}\ y_{2,2}\ \cdots\ y_{2,n_2}\ \to \\
&\qquad\ f\ (g_1\ y_{1,1}\ y_{1,2}\ \cdots\ y_{1,n_1})\ (g_2\ y_{2,1}\ y_{2,2}\ \cdots\ y_{2,n_2})) \\
&\qquad \circledast x_{1,1} \circledast x_{1,2} \circledast \cdots \circledast x_{1,n_1} \\
&\qquad \circledast x_{2,1} \circledast x_{2,2} \circledast \cdots \circledast x_{2,n_2} \\
&\qquad \circledast \cdots \\
&\qquad \circledast (\mathsf{pure}\, g_m \circledast x_{m,1} \circledast x_{m,2} \circledast \cdots \circledast x_{m,n_m}) && \because \text{(Ap-Nest-LemmaB)} \\
&\equiv \quad \cdots \\
&\equiv \quad \mathsf{pure}\,(\lambda\, y_{1,1}\ y_{1,2}\ \cdots\ y_{1,n_1} \\
&\qquad\quad y_{2,1}\ y_{2,2}\ \cdots\ y_{2,n_2} \\
&\qquad\quad \cdots \\
&\qquad\quad y_{m,1}\ y_{m,2}\ \cdots\ y_{m,n_m}\ \to \\
&\qquad\ f\ (g_1\ y_{1,1}\ y_{1,2}\ \cdots\ y_{1,n_1}) \\
&\qquad\ (g_2\ y_{2,1}\ y_{2,2}\ \cdots\ y_{2,n_2}) \\
&\qquad\ \cdots \\
&\qquad\ (g_m\ y_{m,1}\ y_{m,2}\ \cdots\ y_{m,n_m}))
\end{aligned}
$$

$$
\begin{aligned}
&\circledast\ x_{1,1} \circledast x_{1,2} \circledast \cdots \circledast x_{1,n_1} \\
&\circledast\ x_{2,1} \circledast x_{2,2} \circledast \cdots \circledast x_{2,n_2} \\
&\circledast\ \cdots \\
&\circledast\ x_{m,1} \circledast x_{m,2} \circledast \cdots \circledast x_{m,n_m} && \because \text{(Ap-Nest-LemmaB)}
\end{aligned}
$$

証明お疲れ様でした．でも (Ap-Nest) はかなり便利な定理ですから，苦労は報われます．安心してください．

ここまで，(Ap-Nest) の証明には，アプリカティヴ則のうち (Ap-Comp), (Ap-Apply), (Ap-Swap) のみを使ってきました．では逆に，(Ap-Nest) とまだ使っていない (Ap-Id) から，(Ap-Comp), (Ap-Apply), (Ap-Swap) を証明できるでしょうか．実は簡単に出来ます．

— (Ap-Comp) の証明

$$
\begin{aligned}
&\mathsf{pure}\ (\circ) \circledast u \circledast v \circledast w \\
&\equiv\ \mathsf{pure}\ (\circ) \circledast (\mathsf{pure}\ \mathsf{id} \circledast u) \circledast (\mathsf{pure}\ \mathsf{id} \circledast v) \circledast (\mathsf{pure}\ \mathsf{id} \circledast w) && \because \text{(Ap-Id)} \\
&\equiv\ \mathsf{pure}\ (\lambda f\ g\ x\ \to\ (\circ)\ (\mathsf{id}\ f)\ (\mathsf{id}\ g)\ (\mathsf{id}\ x)) \circledast u \circledast v \circledast w && \because \text{(Ap-Nest)} \\
&\equiv\ \mathsf{pure}\ (\lambda f\ g\ x\ \to\ f\ (g\ x)) \circledast u \circledast v \circledast w \\
&\equiv\ \mathsf{pure}\ (\lambda f\ g\ x\ \to\ \mathsf{id}\ f\ (\mathsf{id}\ g\ x)) \circledast u \circledast v \circledast w \\
&\equiv\ \mathsf{pure}\ \mathsf{id} \circledast u \circledast (\mathsf{pure}\ \mathsf{id} \circledast v \circledast w) && \because \text{(Ap-Nest)} \\
&\equiv\ u \circledast (v \circledast w) && \because \text{(Ap-Id)}
\end{aligned}
$$

— (Ap-Apply) の証明

$$
\mathsf{pure}\ f \circledast \mathsf{pure}\ x\ \equiv\ \mathsf{pure}\ (f\ x) \qquad \because \text{(Ap-Nest)}
$$

— (Ap-Swap) の証明

$$
\begin{aligned}
&u \circledast \mathsf{pure}\ y \\
&\equiv\ \mathsf{pure}\ \mathsf{id} \circledast (\mathsf{pure}\ \mathsf{id} \circledast u) \circledast \mathsf{pure}\ y && \because \text{(Ap-Id)} \\
&\equiv\ \mathsf{pure}\ (\lambda x\ \to\ \mathsf{id}\ (\mathsf{id}\ x)\ y) \circledast u && \because \text{(Ap-Apply)} \\
&\equiv\ \mathsf{pure}\ (\lambda x\ \to\ x\ y) \circledast u \\
&\equiv\ \mathsf{pure}\ (\$\ y) \circledast u
\end{aligned}
$$

とにかく，(Ap-Nest) が非常に強力なのです．

以上より，(Ap-Id), (Ap-Comp), (Ap-Apply), (Ap-Swap) の 4 つと (Ap-Id), (Ap-Nest) の 2 つが等価であることが分かりました．このことは重要です．アプリカティヴに関する性質を証明する際は (Ap-Id), (Ap-Nest) を使うのが便利でしょう．ある型がアプリカティヴであることを証明する際は (Ap-Id), (Ap-Comp), (Ap-Apply), (Ap-Swap) を示す方が簡単でしょう．こうしてみると，最初は少し奇妙に見えたアプリカティヴ則も非常に上手く選ばれていることが分かりますね．

以降の証明でも，(Ap-Id), (Ap-Nest) は大活躍します．なお，両者を用いる場合も，(Ap-Id) は省いて，(Ap-Nest) のみ書くことにします．

2 モノイダルとアプリカティヴ

これまでで，アプリカティヴ則が何となく納得できたと思います．でも，それでもまだ気持ち悪さが残っていると思います．おそらくその原因は $\circledast$ が左右の扱いについて非対称であることにあると思います．実はアプリカティヴと等価で対称性の高いモノイダルという型クラスがあるのです．

```
class Functor φ ⇒ Monoidal φ where
  unit :: φ ()
  (⋆) :: φ α → φ β → φ (α, β)    — "zip"
infixl 4 ⋆
```

モノイダル則は以下のとおりです．

結合性 $\mathsf{assocprod}\ \langle\$\rangle\ (u \star (v \star w)) \equiv (u \star v) \star w$ (Zip-Assoc)

左恒等性 $\mathsf{snd}\ \langle\$\rangle\ (\mathsf{unit} \star v) \equiv v$ (Unit-Zip-LIdent)

右恒等性 $\mathsf{fst}\ \langle\$\rangle\ (u \star \mathsf{unit}) \equiv u$ (Unit-Zip-RIdent)

「モノイダル」という名前だけあって，「モノイドっぽい」ですね．アプリカティヴ則に比べれば，モノイダル則はずっと納得できると思います．

これから，アプリカティヴとモノイダルの間の関係を探っていきましょう．

アプリカティヴとモノイダルの間の変換は，型を見れば分かります．アプリカティヴからモノイダルへの変換は，

$\mathsf{unit} \equiv \mathsf{pure}\ ()$ (Pure-Unit)

$fa \star fb \equiv (,)\ \langle\$\rangle\ fa \circledast fb$ (Ap-Zip)

となり，モノイダルからアプリカティヴへの変換は，

$\mathsf{pure}\ x \equiv \mathsf{const}\ x\ \langle\$\rangle\ \mathsf{unit}$ (Unit-Pure)

$mf \circledast mx \equiv (\lambda\,(f, x) \to f\ x)\ \langle\$\rangle\ (mf \star mx)$ (Zip-Ap)

となります．

それでは，アプリカティヴ則とモノイダル則が等価であることを証明していきましょう．

なお，**自然性** より次のことが成立していることに注意してください．

$(f \otimes g)\ \langle\$\rangle\ (u \star v) \equiv (f\ \langle\$\rangle\ u) \star (g\ \langle\$\rangle\ v)$ (Zip-Nat)

まずはモノイダル則からアプリカティヴ則を証明していきます．(Ap-Fmap) を先に証明することにします．

— (Ap-Fmap) の証明

$\mathsf{pure}\ f \circledast x$

$\equiv (\lambda\,(g, y) \to g\ y)\ \langle\$\rangle\ ((\mathsf{const}\ f\ \langle\$\rangle\ \mathsf{unit}) \star x)$ ∵ (Unit-Pure), (Zip-Ap)

$\equiv (\lambda\,(g, y) \to g\ y)\ \langle\$\rangle\ ((\mathsf{const}\ f\ \langle\$\rangle\ \mathsf{unit}) \star (\mathsf{id}\ \langle\$\rangle\ x))$ ∵ (Fmap-Id)

$\equiv (\lambda\,(g, y) \to (\mathsf{const}\ f\ g)\ (\mathsf{id}\ y))\ \langle\$\rangle\ (\mathsf{unit} \star x)$ ∵ (Zip-Nat), (Fmap-Comp)

$\equiv (\lambda\,(\_, y) \to f\ y)\ \langle\$\rangle\ (\mathsf{unit} \star x)$

$\equiv (f \circ \mathsf{snd})\ \langle\$\rangle\ (\mathsf{unit} \star x)$

$\equiv f\ \langle\$\rangle\ (\mathsf{snd}\ \langle\$\rangle\ (\mathsf{unit} \star x))$ ∵ (Fmap-Comp)

$\equiv f\ \langle\$\rangle\ x$ ∵ (Unit-Zip-LIdent)

— (Ap-Id) の証明

$\mathsf{pure}\ \mathsf{id} \circledast x$

$\equiv$ id ⟨$⟩ x ∵ (Ap-Fmap)
$\equiv$ x ∵ (Fmap-Id)

— (Ap-Comp) の証明

pure $(\circ) \circledast u \circledast v \circledast w$
$\equiv$ fmap $(\lambda\,(xy, z) \to xy\ z)$
  ((fmap $(\lambda\,(x, y) \to x\ y)$ (fmap $(\circ)\ u) \star v) \star w$) ∵ (Ap-Fmap), (Unit-Pure), (Zip-Ap)
$\equiv$ fmap $(\lambda\,(xy, z) \to xy\ z)$
  (fmap $(\lambda\,(x, y) \to x \circ y)\ (u \star v) \star w$) ∵ (Fmap-Id), (Zip-Nat), (Fmap-Comp)
$\equiv$ fmap $(\lambda\,((x, y), z) \to (x \circ y)\ z)\ ((u \star v) \star w)$ ∵ (Fmap-Id), (Zip-Nat), (Fmap-Comp)
$\equiv$ fmap $(\lambda\,((x, y), z) \to x\ (y\ z))\ ((u \star v) \star w)$
$\equiv$ fmap $(\lambda\,(x, (y, z)) \to x\ (y\ z))\ (u \star (v \star w))$ ∵ (Zip-Assoc), (Fmap-Comp)
$\equiv$ fmap $(\lambda\,(x, yz) \to x\ yz)\ (u \star$ fmap $(\lambda\,(y, z) \to y\ z)\ (v \star w))$ ∵ (Fmap-Comp), (Zip-Nat), (Fmap-Id)
$\equiv$ $u \circledast (v \circledast w)$ ∵ (Unit-Pure), (Zip-Ap)

— (Ap-Apply) の証明

pure $f \circledast$ pure x
$\equiv$ fmap f (fmap (const x) unit) ∵ (Ap-Fmap), (Unit-Pure)
$\equiv$ fmap (const $(f\ x)$) unit ∵ (Fmap-Comp)
$\equiv$ pure $(f\ x)$ ∵ (Unit-Pure)

— (Ap-Swap) の証明

$u \circledast$ pure y
$\equiv$ fmap $(\lambda\,(f, x) \to f\ x)\ (u \star$ fmap (const y) unit) ∵ (Unit-Pure), (Zip-Ap)
$\equiv$ fmap $(\lambda\,(f, \_) \to f\ y)\ (u \star$ unit) ∵ (Fmap-Id), (Zip-Nat), (Fmap-Comp)
$\equiv$ fmap $((\$\ y) \circ$ snd) $(u \star$ unit)
$\equiv$ fmap $(\$)\ u$ ∵ (Fmap-Comp), (Unit-Zip-RIdent)
$\equiv$ fmap $(\$\ y)\ u$
$\equiv$ pure $(\$\ y) \circledast u$ ∵ (Ap-Fmap)

次は，アプリカティヴ則からモノイダル則を証明していきます．

— (Zip-Assoc) の証明

assocprod ⟨$⟩ $(u \star (v \star w)$
$\equiv$ assocprod ⟨$⟩ $(,)$ ⟨$⟩ $u \circledast ((,)$ ⟨$⟩ $v \circledast w)$ ∵ (Ap-Zip)
$\equiv$ $(\lambda\,x\ y\ z \to$ assocprod $(x, (y, z)))$ ⟨$⟩ $u \circledast v \circledast w$ ∵ (Ap-Nest)
$\equiv$ $(\lambda\,x\ y\ z \to ((x, y), z))$ ⟨$⟩ $u \circledast v \circledast w$
$\equiv$ $(,)$ ⟨$⟩ $((,)$ ⟨$⟩ $u \circledast v) \circledast w$ ∵ (Ap-Nest)
$\equiv$ $(u \star v) \star w$ ∵ (Ap-Zip)

— (Unit-Zip-LIdent) の証明

snd ⟨$⟩ $(\text{unit} \star v)$
$\equiv$ snd ⟨$⟩ $((,)$ ⟨$⟩ (pure $()) \circledast v)$
$\equiv$ $(\lambda\,y \to$ snd $((), y))$ ⟨$⟩ v ∵ (Ap-Nest)
$\equiv$ id ⟨$⟩ v
$\equiv$ v ∵ (Fmap-Id)

— (Unit-Zip-RIdent) の証明

$$
\begin{aligned}
&\mathsf{fst}\ \langle\$\rangle\ (u \star \mathsf{unit}) \\
&\equiv\ \mathsf{fst}\ \langle\$\rangle\ ((,)\ \langle\$\rangle\ v \circledast (\mathsf{pure}\ ())) \\
&\equiv\ (\lambda\, x\ \to\ \mathsf{fst}\ (x, ()))\ \langle\$\rangle\ v && \because \text{(Ap-Nest)} \\
&\equiv\ \mathsf{id}\ \langle\$\rangle\ v \\
&\equiv\ v && \because \text{(Fmap-Id)}
\end{aligned}
$$

実に (Ap-Nest) を使うと実に簡単に証明できてしまいますね．

アプリカティヴ則とモノイダル則の等価性が証明できたので，アプリカティヴにモノイダルを併合して，

$$
\begin{aligned}
&\textbf{class}\ \mathsf{Functor}\ \varphi \Rightarrow \mathsf{Applicative}\ \varphi\ \textbf{where} \\
&\quad \mathsf{pure} :: \alpha\ \to\ \varphi\,\alpha \\
&\quad (\circledast) :: \varphi\,(\alpha\ \to\ \beta)\ \to\ \varphi\,\alpha\ \to\ \varphi\,\beta \quad \text{— “ap”} \\
&\quad \mathsf{unit} :: \varphi\,() \\
&\quad (\star) :: \varphi\,\alpha\ \to\ \varphi\,\beta\ \to\ \varphi\,(\alpha, \beta) \quad \text{— “zip”} \\
&\quad \mathsf{pure}\ x = \mathsf{const}\ x\ \langle\$\rangle\ \mathsf{unit} \\
&\quad mf \circledast mx = (\$)\ \langle\$\rangle\ (mf \star mx) \\
&\quad \mathsf{unit} = \mathsf{pure}\ () \\
&\quad fa \star fb = (,)\ \langle\$\rangle\ fa \circledast fb
\end{aligned}
$$

としてもいいですね．以降この定義を使っていきます．

3 アプリカティヴをゲットだぜ

アプリカティヴについてのさまざまな性質が見えてきましたが，アプリカティヴになるデータ型にはどのようなものがあるのでしょうか．後できちんと証明しますが，モナドになるデータ型は必ずアプリカティヴにすることができます．でも，アプリカティヴはモナドよりずっと簡単に作ることが出来るのです．

まず，アプリカティヴを単純に反転させることで，新たなアプリカティヴを得られます．「反転させる」という言葉の意味は，特に $\star$ に注目すればわかると思います．

$$
\begin{aligned}
&\textbf{newtype}\ \mathsf{OpAp}\ \varphi\,\alpha\ =\ \mathsf{OpAp}\ (\varphi\,\alpha) \\
&\textbf{instance}\ \mathsf{Applicative}\ \varphi \Rightarrow \mathsf{Applicative}\ (\mathsf{OpAp}\ \varphi)\ \textbf{where} \\
&\quad \mathsf{pure}\ x = \mathsf{OpAp}\ (\mathsf{pure}\ x) \\
&\quad \mathsf{OpAp}\ f \circledast \mathsf{OpAp}\ x = \mathsf{OpAp}\ \$\ (\lambda\, y\ g\ \to\ g\ y)\ \langle\$\rangle\ x \circledast f \\
&\quad \mathsf{unit} = \mathsf{OpAp}\ \mathsf{unit} \\
&\quad \mathsf{OpAp}\ x \star \mathsf{OpAp}\ y\ =\ \mathsf{OpAp}\ (\mathsf{swap}\ \langle\$\rangle\ (y \star x))
\end{aligned}
$$

アプリカティヴ則だけ証明しておきます．

— (Ap-Id) の証明

$$
\mathsf{pure}\ \mathsf{id} \circledast \mathsf{OpAp}\ x
$$

$$
\begin{aligned}
&\equiv \mathsf{OpAp}\ \$\ (\lambda\, y\ g\ \to\ g\ y)\ \langle\$\rangle\ x \circledast \mathsf{pure}\ \mathsf{id} \\
&\equiv \mathsf{OpAp}\ \$\ (\lambda\, y\ \to\ \mathsf{id}\ y)\ \langle\$\rangle\ x \\
&\equiv \mathsf{OpAp}\ \$\ \mathsf{id}\ \langle\$\rangle\ x \\
&\equiv \mathsf{OpAp}\ x && \because \text{(Fmap-Id)}
\end{aligned}
$$

— (Ap-Comp) の証明

$$
\begin{aligned}
&\mathsf{pure}\ (\circ) \circledast \mathsf{OpAp}\ u \circledast \mathsf{OpAp}\ v \circledast \mathsf{OpAp}\ w \\
&\equiv \mathsf{OpAp}\ \$\ (\lambda\, z\ cxy\ \to\ cxy\ z)\ \langle\$\rangle\ w \\
&\qquad \circledast ((\lambda\, y\ cx\ \to\ cx\ y)\ \langle\$\rangle\ v \circledast ((\lambda\, x\ c\ \to\ c\ x)\ \langle\$\rangle\ u \circledast \mathsf{pure}\ (\circ))) \\
&\equiv \mathsf{OpAp}\ \$\ (\lambda\, z\ y\ x\ \to\ (x\ \circ\ y)\ z)\ \langle\$\rangle\ w \circledast v \circledast u && \because \text{(Ap-Nest)} \\
&\equiv \mathsf{OpAp}\ \$\ (\lambda\, z\ y\ x\ \to\ x\ (y\ z))\ \langle\$\rangle\ w \circledast v \circledast u \\
&\equiv \mathsf{OpAp}\ \$\ (\lambda\, yz\ x\ \to\ x\ yz)\ \langle\$\rangle\ ((\lambda\, z\ y\ \to\ y\ z)\ \langle\$\rangle\ w \circledast v) \circledast u && \because \text{(Ap-Nest)} \\
&\equiv \mathsf{OpAp}\ u \circledast (\mathsf{OpAp}\ v \circledast \mathsf{OpAp}\ w)
\end{aligned}
$$

— (Ap-Apply) の証明

$$
\begin{aligned}
&\mathsf{pure}\ f \circledast \mathsf{pure}\ x \\
&\equiv \mathsf{OpAp}\ \$\ (\lambda\, y\ g\ \to\ g\ y)\ \langle\$\rangle\ \mathsf{pure}\ x \circledast \mathsf{pure}\ f \\
&\equiv \mathsf{OpAp}\ (\mathsf{pure}\ (f\ x)) && \because \text{(Ap-Nest)} \\
&\equiv \mathsf{pure}\ (f\ x)
\end{aligned}
$$

— (Ap-Swap) の証明

$$
\begin{aligned}
&\mathsf{OpAp}\ u \circledast \mathsf{pure}\ y \\
&\equiv (\lambda\, x\ g\ \to\ g\ x)\ \langle\$\rangle\ \mathsf{pure}\ y \circledast u \\
&\equiv \mathsf{pure}\ (\$\ y) \circledast u && \because \text{(Ap-Nest)}
\end{aligned}
$$

簡単ですね．とはいえ二回反転したら元に戻りますから，これは序の口です．

アプリカティヴをつくるためのもっと強力なメソッドがあります —— アプリカティヴとアプリカティヴを合成すると，なんとまたしてもアプリカティヴができるのです．

```
instance (Applicative φ, Applicative φ') ⇒ Applicative (φ :∘ φ') where
  pure x              = Comp $ pure (pure x)
  Comp f ⊛ Comp x     = Comp $ (⊛) ⟨$⟩ f ⊛ x
  unit                = Comp $ pure unit
  Comp x ⋆ Comp y     = Comp $ (⋆) ⟨$⟩ x ⊛ y
```

これがアプリカティヴ則を満たしていることは，φ における (Ap-Nest) を使い，φ' についてのアプリカティヴ則を引っ張ってくれば，明らかです．これは新しいアプリカティヴを創り出すうえで非常に便利な性質です．

なお，モナドとモナドを単純に合成してもモナドになるとは限りません（だからこそモナド変換子があるわけです）．ですから，このメソッドを使えばモナドでないアプリカティヴも簡単に考えることができます．たとえば，Maybe :∘ State Int などです．

モナドでないアプリカティヴとしてもう少し非自明なものを挙げましょう．

まずは（ちょっとだけ有名な）ZipList です．

```
newtype ZipList α = ZipList [α]
```

```
instance Functor ZipList where
  fmap f (ZipList xs) = ZipList (map f xs)
instance Applicative ZipList where
  pure x                = ZipList (repeat x)
  ZipList fs ⊛ ZipList xs = ZipList $ zipWith ($) fs xs

repeat :: α → [α]
repeat x = x : repeat x

zipWith :: (α → β → γ) → [α] → [β] → [γ]
zipWith f (x : xs) (y : ys) = f x y : zipWith f xs ys
zipWith _ _ _ = []
```

zipWith をさらにモナドにすることはできません.[1] ただし，リストを無限リストに制限した場合は，すんなりモナドに出来ます．これは，Reader モナドと本質的に同じことです．

もう一つが Either モナドの変種です．

```
data Except ε α = OK α | Failed ε
instance Monoid ε ⇒ Applicative (Except ε) where
  pure              = OK
  OK f ⊛ OK x       = OK (f x)
  OK f ⊛ Failed e   = Failed e
  Failed e ⊛ OK x   = Failed e
  Failed e ⊛ Failed e' = Failed (e ◇ e')
```

これは，Either モナドと違い，エラーを蓄積していくことができます．アプリカティヴはモナドと違い，（表現しにくいですが）値の相互の依存度がとても小さいので，このような芸当も簡単にできるのです．

4 アプリカティヴをバリバリ使おう

アプリカティブはさまざまな所で役に立っています．たとえば Parsec などのモナディックパーサーを使う際も，⊛ を用いてソースコードを簡潔にする「アプリカティヴスタイル」は頻繁に使うと思います．それ以外にも，いろいろな使い方があります．

非常に基本的な型クラスとして，フォルダブルとトラヴァーサブルがあります．

```
class Foldable τ where
  foldMap :: Monoid μ ⇒ (α → μ) → τ α → μ
fold :: Monoid μ ⇒ τ μ → μ
fold = foldMap id
class (Functor τ, Foldable τ) ⇒ Traversable τ where
  sequenceA :: Applicative φ ⇒ τ (φ α) → φ (α)
```

[1] もちろん，リストモナドは作れますが，ここで定義した pure と ⊛ を拡張した形では（つまり (Ret-Pure) と (Bind-Ap) を満たす形では），モナドのインスタンスに出来ないということです．

```
traverse     :: Applicative φ ⇒ (α → φ β) → τ α → φ (τ β)
traverse f x = sequenceA (fmap f x)
```

トラヴァーサブル則などはここではおいておくことにして，とりあえず Tree を Traversable のインスタンスにしていきましょう．

```
instance Foldable Tree where
  foldMap f Leaf         = ∅
  foldMap f (Node l x r) = foldMap f l ◇ f x ◇ foldMap f r
instance Traversable Tree where
  sequenceA Leaf         = pure Leaf
  sequenceA (Node l x r) = Node ⟨$⟩ sequenceA l ⊛ x ⊛ sequenceA r
```

雰囲気は掴めたと思います．かなり単純な手順ですね．実際，GHC 拡張の DeriveFoldable や DeriveTraversable を使えば，この種のインスタンスを自動的に作らせることができます．フォルダブルではモノイドでデータをくっつけていきますが，トラヴァーサブルではアプリカティヴを使って構造を保ちつつデータをくっつけていく，という感じです．

アローを考えるときにもアプリカティヴは活躍します．まずは，アローの上位クラスであるカテゴリーと，アローを定義しましょう．

```
class Category α where
  idA   :: α β β
  (⋙) :: Category α ⇒ α β γ → α γ δ → α β δ
class Category α ⇒ Arrow α where
  arr   :: (β → γ) → α β γ
  first :: α β γ → α (β, δ) (γ, δ)
returnA :: Arrow α ⇒ α β β
returnA = arr id
```

カテゴリー則は

結合性	$(f \ggg g) \ggg h \equiv f \ggg (g \ggg h)$	(Cat-Assoc)
左恒等性	$\mathsf{idA} \ggg f \equiv f$	(Cat-LIdent)
右恒等性	$f \ggg \mathsf{idA} \equiv f$	(Cat-RIdent)

であり，そこに加わるアロー則は

関手恒等性	$\mathsf{arr}\ \mathsf{id} \equiv \mathsf{idA}$	(Arr-Id)
関手合成	$\mathsf{arr}\ (g \circ f) \equiv \mathsf{arr}\ f \ggg \mathsf{arr}\ g$	(Arr-Comp)
拡張	$\mathsf{first}\ (\mathsf{arr}\ f) \equiv \mathsf{arr}\ (f \otimes \mathsf{id})$	(First-Extension)
関手	$\mathsf{first}\ (f \ggg g) \equiv \mathsf{first}\ f \ggg \mathsf{first}\ g$	(First-Functor)
交換	$\mathsf{first}\ f \ggg \mathsf{arr}\ (\mathsf{id} \otimes g) \equiv \mathsf{arr}\ (\mathsf{id} \otimes g) \ggg \mathsf{first}\ f$	(First-Swap)
単位	$\mathsf{first}\ f \ggg \mathsf{arr}\ \mathsf{fst} \equiv \mathsf{arr}\ \mathsf{fst} \ggg f$	(First-Unit)

結合　　first (first f) ⋙ arr assocprod ≡ arr assocprod ⋙ first f　　(First-Assoc)

です．結構多いですね．とりあえずよく分からなくても構いません．

さて，カテゴリーとアプリカティヴがあれば新たなカテゴリーが得られ，アローとアプリカティヴがあれば新たなアローが得られます．

```
newtype Static φ α β γ = Static (φ (α β γ))
instance (Applicative φ, Category α) ⇒ Category (Static φ α) where
  idA                 = Static $ pure idA
  Static f ⋙ Static g = Static $ (⋙) <$> f <*> g
instance (Applicative φ, Arrow α) ⇒ Arrow (Static φ α) where
  arr f               = Static $ pure (arr f)
  first (Static f) = Static $ first <$> f
```

それぞれがきちんとカテゴリー則やアロー則を満たしていることは (Ap-Nest) によってすぐに証明できます．アプリカティヴはたくさん作れますから，この方法を使えば，カテゴリーやアローもたくさん作れますね．

第五章

モナド

1　モナドの基本

モナドについての文章は掃いて捨てるほどあります．モナドは確かに Haskell 入門における一つの関門でしょう．しかし，モナドも数多くある型クラスの一つに過ぎません．気負わず気楽に考えていきましょう．

まずは，モナドといくつかの関数を定義しましょう．

```
class Functor μ ⇒ Monad (μ :: * → *) where
  return :: α → μ α
  (▷)    :: μ α → (α → μ β) → μ β    — "Bind"
(≫)    :: Monad μ ⇒ μ α → μ β → μ β    — "Then"
x ≫ y = x ▷ const y
(◊)    :: Monad μ ⇒ (α → μ β) → (β → μ γ) → (α → μ γ)    — "Kleisli (クライスリ) Composition"
f ◊ g  = λ a → f a ▷ g
join   :: Monad μ ⇒ μ (μ α) → μ α
join m = m ▷ id
infixl 1 ▷, ≫
infixr 1 ◊
```

◊ や join は見慣れないかもしれません．定義をしっかり確認しておいてください．

2　モナド則

一般的なモナド則は以下の三つです．

結合性	$m \triangleright \lambda\, m' \to k\, m' \triangleright h \equiv (m \triangleright k) \triangleright h$	(Bind-Assoc)
左恒等性	return $x \triangleright k \equiv k\, x$	(Bind-Ret-LIdent)
右恒等性	$m \triangleright$ return $\equiv m$	(Bind-Ret-RIdent)

これは見慣れているでしょうが，何となく気持ち悪さが残ると思います．その原因は，▷ が左右について非対称であることにあると思います．

悲しいことに，多くの人は最初にこのモナド則に触れ合い，もどかしさを抱えるようです．圏論を学

べばモナド則もよく分かる，といった話もよく聞きますが，ここではもう少し親しみやすい形で説明していきます．

モナド則は少し工夫をするだけで見違えるほど美しくなります．次のようにすればいいのです．

結合性	$a \diamondsuit (b \diamondsuit c) \equiv (a \diamondsuit b) \diamondsuit c$	(Klei-Assoc)
左恒等性	$\mathsf{return} \diamondsuit a \equiv a$	(Klei-Ret-LIdent)
右恒等性	$a \diamondsuit \mathsf{return} \equiv a$	(Klei-Ret-RIdent)

なんだかモノイドそっくりですね．でも，モノイドと違って（表現しづらいですが）型の幅があります．これがまさに圏なのです．モナドが成している圏は**クライスリ圏** (Kleisli category) といいます．とはいえ，ここでは圏論の話はどうでもいいのです．

$\triangleright$ 版モナド則と $\diamondsuit$ 版モナド則が等価であることを証明しましょう．三つがそれぞれ対応しているので，分かりやすいです．

— (Bind-Assoc) $\iff$ (Klei-Assoc) の証明

$$
\begin{aligned}
&\forall\, m.\, (m \rhd f) \rhd g \equiv m \rhd \lambda\, m' \to f\ m' \rhd g \\
&\iff \forall\, k.\, \lambda\, x \to (k\ x \rhd f) \rhd g \equiv \lambda\, x \to k\ x \rhd \lambda\, m' \to f\ m' \rhd g \\
&\iff \forall\, k.\, (k \diamondsuit f) \diamondsuit g \equiv k \diamondsuit (f \diamondsuit g)
\end{aligned}
$$

— (Bind-Ret-LIdent) $\iff$ (Klei-Ret-LIdent) の証明

$$
\begin{aligned}
&\forall\, x.\, \mathsf{return}\ x \rhd f \equiv f\ x \\
&\iff \forall\, x.\, \lambda\, x \to \mathsf{return}\ x \rhd f \equiv f \\
&\iff \mathsf{return} \diamondsuit f \equiv f
\end{aligned}
$$

— (Bind-Ret-RIdent) $\iff$ (Klei-Ret-RIdent) の証明

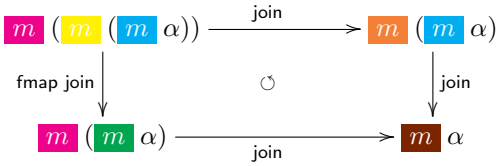
$$
\begin{aligned}
&\forall\, m.\, m \rhd \mathsf{return} \equiv m \\
&\iff \forall\, f.\, f\ x \rhd \mathsf{return} \equiv f \\
&\iff f \diamondsuit \mathsf{return} \equiv f
\end{aligned}
$$

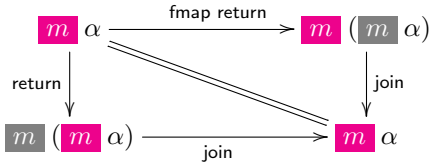
細かい部分は省略しています．興味があれば省略した部分の論理を考えてみてください．

モナド則の別のバージョンとして，join を使うものもあります．

結合性	$\mathsf{join} \circ \mathsf{join} \equiv \mathsf{join} \circ \mathsf{fmap}\ \mathsf{join}$	(Join-Assoc)
左恒等性	$\mathsf{join} \circ \mathsf{return} \equiv \mathsf{id}$	(Join-Ret-LIdent)
右恒等性	$\mathsf{join} \circ \mathsf{fmap}\ \mathsf{return} \equiv \mathsf{id}$	(Join-Ret-RIdent)

式だけ見ていると分かりにくいかもしれませんね．図にするとこんな感じです．色を見ればどの部分を操作しているか分かります．





これはこれで結構美しいですよね．

▷ 版モナド則と join 版モナド則はもちろん等価です．少し手間がかかりますが，等価であることを証明しましょう．

先にいくつかの関係をまとめておきます．▷ と return から join と ⟨$⟩ が作れます．

$$\mathsf{join} \equiv (\triangleright\ \mathsf{id}) \qquad \text{(Bind-Join)}$$
$$f\ \langle\$\rangle\ m \equiv m \triangleright f\ \mathsf{return} \qquad \text{(Bind-Ret-Fmap)}$$

そして，join と ⟨$⟩ から ▷ が作れます．

$$m \triangleright f \equiv \mathsf{join}\ (f\ \langle\$\rangle\ m) \qquad \text{(Join-Fmap-Bind)}$$

きちんと型が対応していることを確認してください．

さらに，**自然性**より次の二つが成立しています．

$$\mathsf{fmap}\ f \circ \mathsf{return} \equiv \mathsf{return} \circ f \qquad \text{(Ret-Nat)}$$
$$\mathsf{fmap}\ f \circ \mathsf{join} \equiv \mathsf{join} \circ \mathsf{fmap}\ (\mathsf{fmap}\ f) \qquad \text{(Join-Nat)}$$

この二つが証明のポイントです．ここはよく確認してください．

では，▷ 版モナド則から join 版モナド則を導きましょう．

— (Join-Assoc) の証明

$(\mathsf{join} \circ \mathsf{join})\ m$
$\equiv (m \triangleright \mathsf{id}) \triangleright \mathsf{id}$ ∵ (Bind-Join)
$\equiv m \triangleright \lambda\, m' \to m' \triangleright \mathsf{id}$ ∵ (Bind-Assoc)
$(\mathsf{join} \circ \mathsf{fmap}\ \mathsf{join})\ m$
$\equiv (m \triangleright \lambda\, m' \to \mathsf{return}\ (\mathsf{join}\ m')) \triangleright \mathsf{id}$ ∵ (Bind-Ret-Fmap), (Bind-Join)
$\equiv m \triangleright \lambda\, y \to y \triangleright \mathsf{id}$ ∵ (Bind-Assoc), (Bind-Ret-LIdent)
∴ $\mathsf{join} \circ \mathsf{join} \equiv \mathsf{join} \circ \mathsf{fmap}\ \mathsf{join}$

— (Join-Ret-LIdent) の証明

$(\mathsf{join} \circ \mathsf{return})\ m$
$\equiv \mathsf{return}\ m \triangleright \mathsf{id}$ ∵ (Bind-Join)
$\equiv m$ ∵ (Bind-Ret-RIdent)
∴ $\mathsf{join} \circ \mathsf{return} \equiv \mathsf{id}$

— (Join-Ret-RIdent) の証明

$(\mathsf{join} \circ \mathsf{fmap}\ \mathsf{return})\ m$
$\equiv m \triangleright (\lambda\, m' \to \mathsf{return}\ (\mathsf{return}\ m')) \triangleright \mathsf{id}$ ∵ (Bind-Join), (Bind-Ret-Fmap)
$\equiv m \triangleright \lambda\, m' \to \mathsf{return}\ (\mathsf{return}\ m') \triangleright \mathsf{id}$ ∵ (Bind-Assoc)
$\equiv m \triangleright \mathsf{return}$ ∵ (Bind-Ret-LIdent)
$\equiv m$ ∵ (Bind-Ret-RIdent)
∴ $\mathsf{join} \circ \mathsf{fmap}\ \mathsf{return} \equiv \mathsf{id}$

次は，join 版モナド則から ▷ 版モナド則を導きましょう．

— (Bind-Assoc) の証明

$$
\begin{aligned}
&(m \rhd f) \rhd g \\
&\equiv (\mathsf{join} \circ \mathsf{fmap}\ g \circ \mathsf{join} \circ \mathsf{fmap}\ f)\ m && \because \text{(Join-Fmap-Bind)} \\
&m \rhd \lambda\, m' \to f\ m' \rhd g \\
&\equiv \mathsf{join}\ ((\lambda\, m' \to \mathsf{join}\ (\mathsf{fmap}\ g\ (f\ m')))\ \langle\$\rangle\ m) && \because \text{(Join-Fmap-Bind)} \\
&\equiv (\mathsf{join} \circ \mathsf{fmap}\ (\mathsf{join} \circ \mathsf{fmap}\ g \circ f))\ m \\
&\equiv (\mathsf{join} \circ \mathsf{fmap}\ \mathsf{join} \circ \mathsf{fmap}\ (\mathsf{fmap}\ g) \circ \mathsf{fmap}\ f)\ m && \because \text{(Fmap-Comp)} \\
&\equiv (\mathsf{join} \circ \mathsf{join} \circ \mathsf{fmap}\ (\mathsf{fmap}\ g) \circ \mathsf{fmap}\ f)\ m && \because \text{(Join-Assoc)} \\
&\equiv (\mathsf{join} \circ \mathsf{fmap}\ g \circ \mathsf{join} \circ \mathsf{fmap}\ f)\ m && \because \text{(Join-Nat)} \\
&\therefore (m \rhd f) \rhd g \equiv m \rhd \lambda\, m' \to f\ m' \rhd g
\end{aligned}
$$

— (Bind-Ret-LIdent) の証明

$$
\begin{aligned}
&\mathsf{return}\ x \rhd f \\
&\equiv \mathsf{join}\ (\mathsf{fmap}\ f\ (\mathsf{return}\ x)) && \because \text{(Join-Fmap-Bind)} \\
&\equiv \mathsf{join}\ (\mathsf{return}\ (f\ x)) && \because \text{(Ret-Nat)} \\
&\equiv f\ x && \because \text{(Join-Ret-LIdent)}
\end{aligned}
$$

— (Bind-Ret-RIdent) の証明

$$
\begin{aligned}
&m \rhd \mathsf{return} \\
&\equiv \mathsf{join}\ (\mathsf{fmap}\ \mathsf{return}\ m) && \because \text{(Join-Fmap-Bind)} \\
&\equiv m && \because \text{(Join-Ret-LIdent)}
\end{aligned}
$$

証明お疲れ様でした．この証明を通して，モナド則がどういうものであるかはよくわかったと思います．

さて，$\rhd$ 版モナド則と join 版モナド則が等価であることを踏まえると，モナドの定義に join を加えて以下のようにしてもいいでしょう．

```
class Functor μ ⇒ Monad (μ :: * → *) where
  return :: α → μ α
  (▷)    :: μ α → (α → μ β) → μ β
  join   :: μ (μ α) → μ α

  x ▷ f  = join (f ⟨$⟩ x)
  join x = x ▷ id
```

join は新たなモナドを考える際にも便利です．$\rhd$ の型は $\mu\,\alpha \to (\alpha \to \mu\,\beta) \to \mu\,\beta$ であり，$\rhd$ を実装しようと思うと二引数がかかわるのでちょっとややこしいですが，join の型は $\mu\,(\mu\,\alpha) \to \mu\,\alpha$ であり，一引数しか考えなくてよくて単純ですから実装しやすいことが多いのです．もちろん，return もしっかり考えなければなりませんが．

たとえばリストモナドの場合，join は $[[\alpha]] \to [\alpha]$ の型を持ち，return は $\alpha \to [\alpha]$ の型を持ちます．こうして考えると見通しも立ちやすいでしょう．リストモナドの場合，join が concat となり，return が (: []) となります．

ぜひ新しいモナドを考えてみてくださいね．

3 アプリカティヴとモナド

有名な話ですが，モナドはみなアプリカティヴです．次のようにすると，$\triangleright$ と return から pure と $\circledast$ が作れます．

$$\mathsf{pure} \equiv \mathsf{return} \qquad \text{(Ret-Pure)}$$
$$f \circledast x \equiv f \triangleright \lambda f' \to x \triangleright \mathsf{return} \circ f' \qquad \text{(Bind-Ap)}$$

これでアプリカティヴを満たせることをモナド則を使って証明しましょう．

— (Ap-Id) の証明

$$\begin{aligned}
&\mathsf{pure}\ \mathsf{id} \circledast x \\
&\equiv \mathsf{return}\ \mathsf{id} \triangleright \lambda f' \to x \triangleright \mathsf{return} \circ f' && \because \text{(Ret-Pure)} \\
&\equiv x && \because \text{(Ret-Bind-LIdent), (Ret-Bind-RIdent)}
\end{aligned}$$

— (Ap-Comp) の証明

$$\begin{aligned}
&\mathsf{pure}\ (\circ) \circledast u \circledast v \circledast w \\
&\equiv u \triangleright \lambda u' \to \mathsf{return}\ (u' \circ) \\
&\quad \triangleright \lambda f \to v \triangleright \lambda v' \to \mathsf{return}\ (f\ v') \\
&\quad \triangleright \lambda g \to w \triangleright \lambda w' \to \mathsf{return}\ (g\ w') && \because \text{(Bind-Ap), (Bind-Assoc)} \\
&\equiv u \triangleright \lambda u' \to v \triangleright \lambda v' \to \\
&\quad w \triangleright \lambda w' \to \mathsf{return}\ (u'\ (v'\ w')) && \because \text{(Bind-Ret-LIdent)} \\
&\equiv u \triangleright \lambda u' \to v \triangleright \lambda v' \to \\
&\quad w \triangleright \lambda w' \to \mathsf{return}\ (v'\ w') \triangleright \mathsf{return} \circ u' && \because \text{(Bind-Ret-LIdent)} \\
&u \circledast (v \circledast w) && \because \text{(Bind-Ap), (Bind-Assoc)}
\end{aligned}$$

— (Ap-Apply) の証明

$$\begin{aligned}
&\mathsf{pure}\ f \circledast \mathsf{pure}\ x \\
&\equiv \mathsf{return}\ x \triangleright \mathsf{return} \circ f \\
&\equiv \mathsf{return}\ (f\ x) && \because \text{(Bind-Ret-LIdent)} \\
&\equiv \mathsf{pure}\ (f\ x) && \because \text{(Ret-Pure)}
\end{aligned}$$

— (Ap-Swap) の証明

$$\begin{aligned}
&u \circledast \mathsf{pure}\ y \\
&\equiv u \triangleright \lambda u' \to \mathsf{return}\ (u'\ y) && \because \text{(Ret-Pure), (Bind-Ap), (Bind-Ret-LIdent)} \\
&\equiv u \triangleright \mathsf{return} \circ (\$\ y) \\
&\equiv (\$\ y) \circledast u && \because \text{(Bind-Ap)}
\end{aligned}$$

以上を踏まえると，モナドをアプリカティヴの部分クラスにすることもできます．

```
class Applicative μ ⇒ Monad μ where
  return :: α → μ α
  (▷)    :: μ α → (α → μ β) → μ β
  return = pure
```

4 自由モナド

いろいろなモナドがありますが，その中でも重要なのが自由モナドでしょう．

```
data Free (φ :: * → *) α = Pure α
                         | Free (φ (Free φ α))

instance Functor φ ⇒ Functor (Free φ) where
  fmap f (Pure x) = Pure (f x)
  fmap f (Free s) = Free (fmap f ⟨$⟩ s)

instance Functor φ ⇒ Monad (Free φ) where
  return      = Pure
  Pure x ▷ f  = f x
  Free s ▷ f  = Free ((▷ f) ⟨$⟩ s)
```

自由モナドは一般性が高いので，これがモナド則を満たすことの証明は少々頭を使います．ここでは，「レベル」を考え，レベルに関する帰納法を使って証明していきます．「レベル」は，ここでは Free の入れ子の深さを表すとします．例えば Free [] String 型の場合，

```
Pure "foo" ⟹ レベル 0
Free [Pure "this", Free [Free [Pure "is", Pure "a"], Pure "data"]] ⟹ レベル 3
```

となります．

それでは，自由モナドがモナド則を満たすことを証明していきましょう．

— (Bind-Assoc) のレベルに関する帰納法による証明

(a) Pure x はレベル 0 であり，

$$\text{Pure}\ x \rhd f \rhd g \equiv f\ x \rhd g$$

(b) 任意のレベル k 未満の m について $m \rhd f \rhd g \equiv m \rhd \lambda x \to f\ x \rhd g$ が成り立っているとき，Free s がレベル k だとすると，

$$\begin{aligned}
\text{Free}\ s \rhd f \rhd g &\equiv \text{Free}\ ((\rhd g) \langle\$\rangle\ ((\rhd f) \langle\$\rangle\ s)) \\
&\equiv \text{Free}\ ((\lambda m \to m \rhd f \rhd g) \langle\$\rangle\ s) \qquad \because \text{(Fmap-Comp)} \\
\text{Free}\ s \rhd \lambda x \to f\ x \rhd g &\equiv \text{Free}\ ((\lambda m \to m \rhd \lambda x \to f\ x \rhd g) \langle\$\rangle\ s)
\end{aligned}$$

$s :: \varphi\ (\text{Free}\ \varphi\ \alpha)$ を構成している Free $\varphi\ \alpha$ 型のデータはみなレベル k 未満であるので，

$$\lambda m \to m \rhd f \rhd g \langle\$\rangle\ s \equiv (\lambda m \to m \rhd \lambda x \to f\ x \rhd g) \langle\$\rangle\ s$$

ゆえに，

$$\text{Free}\ s \rhd f \rhd g \equiv \text{Free}\ s \rhd \lambda x \to f\ x \rhd g$$

— (Bind-Ret-LIdent) の証明

$$\text{return}\ x \rhd f \equiv \text{Pure}\ x \rhd f \equiv f\ x$$

— (Bind-Ret-RIdent) のレベルに関する帰納法による証明

(a) Pure x はレベル 0 であり，

$$\mathsf{Pure}\ x \rhd \mathsf{return} \equiv \mathsf{return}\ x \equiv \mathsf{Pure}\ x$$

(b) 任意のレベル k 未満の m について $m \rhd \mathsf{return} \equiv m$ が成り立っているとき，
Free s がレベル k だとすると，

$$\mathsf{Free}\ s \rhd \mathsf{return} \equiv \mathsf{Free}\ ((\lambda\, m \to m \rhd \mathsf{return})\ \langle\$\rangle\ s) \equiv \mathsf{Free}\ (\mathsf{id}\ \langle\$\rangle\ s) \equiv \mathsf{Free}\ s$$

めでたく証明が終わりました.

参考文献

[1] Richard Bird. *Introduction to Functional Programming using Haskell, 2nd Edition*. Bell & Bain Ltd, 1988. 山下信夫 訳.『関数プログラミング入門 —Haskell で学ぶ原理と技法』オーム社，2012.

[2] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, Vol. 39, pp. 135–154, 1985.

[3] Jeremy Gibbons and Oege de Moor. *the fun of programming*. Palgrave Macmillan, 2003. 山下伸夫 訳.『関数プログラミングの楽しみ』オーム社，2010.

[4] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. 山本和彦 訳.『プログラミング Haskell』オーム社，2009.

[5] Patricia Johann and Janis Voigtländer. The impact of seq on free theorems-based program transformations. *Fundamenta Informaticae*, 2006.

[6] Miran Lipovača. *Learn You a Haskell for Great Good!: A Beginner's Guide*. no starch press, 2011. オンライン版 http://learnyouahaskell.com/. 田中英行・村主崇行共訳.『すごい Haskell たのしく学ぼう！』オーム社，2012.

[7] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, 2013. オンライン版 http://chimera.labs.oreilly.com/books/1230000000929/index.html.

[8] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 2008.

[9] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, 2009. オンライン版 http://book.realworldhaskell.org/. 山下信夫・伊東勝利・株式会社タイムインターメディア 訳.『Real World Haskell — 実戦で学ぶ関数型言語プログラミング』オーム社，2009.

[10] Benjamin C. Pierce. *Types and Programming Languages*. Massachusetts Institute of Technology, 2002. 住井英二郎監訳. 遠藤侑介・酒井政裕・今井敬吾・黒木裕介・今井宜洋・才川隆文・今井健男訳.『型システム入門 — プログラミング言語と型の理論』オーム社，2013.

[11] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. Massachusetts Institute of Technology, 2005.

[12] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pp. 347–359, 1989.

[13] Philip Wadler. Recursive types for free! http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt, 1990.

索引