9

Function Types

So far I've been glossing over the meaning of function types. A function type is different from other types.

Take `Integer`, for instance: It's just a set of integers. `Bool` is a two element set. But a function type $a \to b$ is more than that: it's a set of morphisms between objects a and b. A set of morphisms between two objects in any category is called a hom-set. It just so happens that in the category **Set** every hom-set is itself an object in the same category—because it is, after all, a *set*.

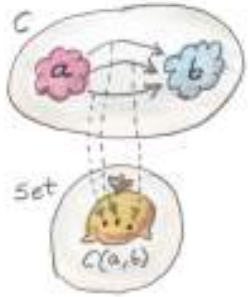
9

Function Types

これまで、関数型の意味について説明してきました。関数型は他の型とは異なります。例えば、Integerは単なる整数の集合です。これは単なる整数の集合です。Bool は2つの要素からなる集合です。しかし、関数型 Ll_lD44E → はそれ以上のもので、オブジェクト Ll_lD44E との間のモルヒズムのセットなのです。任意のカテゴリにおける2つのオブジェクト間の形態素のセットは、hom-setと呼ばれます。たまたま、 ☒というカテゴリでは、すべての同相集合はそれ自体が同じカテゴリのオブジェクトである - なぜなら、結局のところ集合だからである。

Hom-set in Set is just a set

The same is not true of other categories where hom-sets are external to a category. They are even called *external* hom-sets.

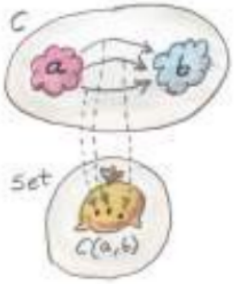


Hom-set in category C is an external set

It's the self-referential nature of the category **Set** that makes function types special. But there is a way, at least in some categories, to construct objects that represent hom-sets. Such objects are called *internal* hom-sets.

Hom-set in Set is just a set

他のカテゴリーでは、hom-set はカテゴリーの外部にある。これらは外的同次集合と呼ばれる。



カテゴリCのhom-setは外部集合である

カテゴリ□　の自己言及的な性質があるからこそ、関数型は特別なのです。しかし、少なくともいくつかのカテゴリでは、ホムセットを表すオブジェクトを構成する方法がある。そのようなオブジェクトは内部ホムセットと呼ばれる。

9.1 Universal Construction

Let's forget for a moment that function types are sets and try to construct a function type, or more generally, an internal hom-set, from scratch. As usual, we'll take our cues from the **Set** category, but carefully avoid using any properties of sets, so that the construction will automatically work for other categories.

A function type may be considered a composite type because of its relationship to the argument type and the result type. We've already seen the constructions of composite types — those that involved relationships between objects. We used universal constructions to define a product and coproduct types. We can use the same trick to define a function type. We will need a pattern that involves three objects: the function type that we are constructing, the argument type, and the result type.

The obvious pattern that connects these three types is called *function application* or *evaluation*. Given a candidate for a function type, let's call it z (notice that, if we are not in the category **Set**, this is just an object like any other object), and the argument type a (an object), the application maps this pair to the result type b (an object). We have three objects, two of them fixed (the ones representing the argument type and the result type).

We also have the application, which is a mapping. How do we incorporate this mapping into our pattern? If we were allowed to look inside objects, we could pair a function f (an element of z) with an argument x (an element of a) and map it to fx (the application of f to x, which is an element of b).

9.1 Universal Construction

関数型が集合であることを少し忘れて、関数型、より一般的には内部ホムセッ トをゼロから構築してみましょう。いつものように、 □カテゴリーからヒントを得ますが、集合の特性を使わないように注意します。関数型は、引数型と結果型との関係から複合型とみなされることがあります。複合型の構成はすでに見てきたとおり、オブジェクト間の関係に関わるものです。普遍構文を使って、積型と共積型を定義しました。同じ方法で関数型も定義することができます。構成する関数型、引数型、結果型という3つのオブジェクトを含むパターンが必要です。この3つの型をつなぐ明らかなパターンが、関数の適用あるいは評価と呼ばれるものである。関数型の候補を□（□カテゴリでなければ、これは他のオブジェクトと同様に単なるオブジェクトであることに注意してください）と引数型 （オブジェクト）を与えると、アプリケーションはこのペアを結果型 （オブジェクト）にマッピングするのです。3つのオブジェクトがあり、そのうち2つは固定されています（引数の型と結果の型を表すもの）。また、マッピングであるアプリケーションもある。このマッピングをどのようにパターンに取り込めばよいのだろうか？もしオブジェクトの中を見ることが許されるなら、関数 □（□の要素）と引数 □（□の要素）を対にして □ □ にマッピングできます（□の要素である □ への □ の application）。

In Set we can pick a function f from a set of functions z and we can pick an argument x from the set (type) a. We get an element fx in the set (type) b.

But instead of dealing with individual pairs (f, x), we can as well talk about the whole *product* of the function type z and the argument type a. The product $z \times a$ is an object, and we can pick, as our application morphism, an arrow g from that object to b. In **Set**, g would be the function that maps every pair (f, x) to fx.

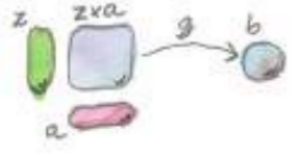
So that's the pattern: a product of two objects z and a connected to another object b by a morphism g.

A pattern of objects and morphisms that is the starting point of the universal construction

Is this pattern specific enough to single out the function type using a universal construction? Not in every category. But in the categories of interest to us it is. And another question: Would it be possible to define a function object without first defining a product? There are categories in

Setでは、関数の集合　から関数⬚を選ぶことができ、集合（型）⬚から引数⬚を選ぶことができる。集合(型)⬚の要素⬚が得られます。

しかし、個々のペア（⬚，　L1_1D465）を扱う代わりに、関数型 ⬚と引数型　　の積全体について話すことができます。積⬚×　L1_1D44E はオブジェクトであり、そのオブジェクトから⬚への矢印⬚を適用形態として選ぶことができる。⬚では、　はすべてのペア（⬚，⬚）を L1_1D465 にマップする関数になります。つまり、2つのオブジェクト⬚と⬚の積が、モルヒズム⬚によって別のオブジェクト⬚に接続されるというパターンである。



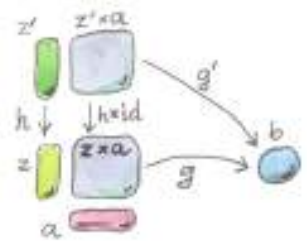
普遍構文の起点となる、オブジェクトとモルヒズムのパターン

このパターンは普遍構文を使って関数型を単一化するのに十分なほど特殊なのだろうか？すべてのカテゴリにおいてではない。しかし、私たちの興味のあるカテゴリではそうである。そしてもう一つの疑問。最初に製品を定義することなく、機能オブジェクトを定義することは可能でしょうか？にはカテゴリーがあります。

which there is no product, or there isn't a product for all pairs of objects. The answer is no: there is no function type, if there is no product type. We'll come back to this later when we talk about exponentials.

Let's review the universal construction. We start with a pattern of objects and morphisms. That's our imprecise query, and it usually yields lots and lots of hits. In particular, in **Set**, pretty much everything is connected to everything. We can take any object z, form its product with a, and there's going to be a function from it to b (except when b is an empty set).

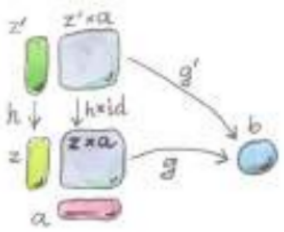
That's when we apply our secret weapon: ranking. This is usually done by requiring that there be a unique mapping between candidate objects — a mapping that somehow factorizes our construction. In our case, we'll decree that z together with the morphism g from $z \times a$ to b is *better* than some other z' with its own application g', if and only if there is a unique mapping h from z' to z such that the application of g' factors through the application of g. (Hint: Read this sentence while looking at the picture.)



Establishing a ranking between candidates for the function object

Now here's the tricky part, and the main reason I postponed this particular universal construction till now. Given the morphism $h :: z' \to z$,

には、積がない、あるいはすべてのペアのオブジェクトに積がない、というカテゴリがあります。答えはノーです。積の型がなければ、関数の型もないのです。このことは、後で指数関数について説明するときに、また触れることにします。普遍構文について復習しておきましょう。まず、オブジェクトとモルヒズムのパターンから始めます。これは我々の不正確な問い合わせであり、通常、たくさんのヒットを生み出す。特に、□□では、ほとんどすべてのものがすべてのものとつながっています。任意のオブジェクト　を取り出し、　と積を形成し、そこからń への関数が存在する（　が空集合である場合を除く）。このとき、秘密兵器である「順位付け」を行う。これは通常、候補となるオブジェクトの間に一意的なマッピングがあることを要求することで行われる。この場合、□と□からの形態素は、他の□×□とその応用例□より優れていると判断する。□から□への写像　Ll_210E が存在し、□の適用が□の適用を分解する場合に限る。（ヒント：この文章は絵を見ながら読んでください）



関数オブジェクトの候補間の順位を確立する

さて、ここが厄介なところで、この特殊な普遍構文を今まで先延ばしにしてきた主な理由でもある。形態素　′　→　□　が与えられたとき。

we want to close the diagram that has both z' and z crossed with a. What we really need, given the mapping h from z' to z, is a mapping from $z' \times a$ to $z \times a$. And now, after discussing the functoriality of the product, we know how to do it. Because the product itself is a functor (more precisely an endo-bi-functor), it's possible to lift pairs of morphisms. In other words, we can define not only products of objects but also products of morphisms.

Since we are not touching the second component of the product $z' \times a$, we will lift the pair of morphisms $(h, \mathbf{id})$, where **id** is an identity on a.

So, here's how we can factor one application, g, out of another application g':

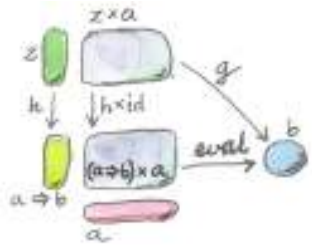
$$g' = g \circ (h \times \mathbf{id})$$

The key here is the action of the product on morphisms.

The third part of the universal construction is selecting the object that is universally the best. Let's call this object $a \Rightarrow b$ (think of this as a symbolic name for one object, not to be confused with a Haskell typeclass constraint — I'll discuss different ways of naming it later). This object comes with its own application — a morphism from $(a \Rightarrow b) \times a$ to b — which we will call *eval*. The object $a \Rightarrow b$ is the best if any other candidate for a function object can be uniquely mapped to it in such a way that its application morphism g factorizes through *eval*. This object is better than any other object according to our ranking.

□と□の両方が□と交差している図を閉じたいのです。□から□への写像 L1_210E が与えられたときに本当に必要なのは、□×□から□× L1_1D44Eへの写像である。そして今、積のfunctorialityを議論した結果、その方法がわかった。積自体がファンクタ（より正確にはエンドバイファンクタ）なので、モルヒズムのペアを持ち上げることができるのです。つまり、オブジェクトの積だけでなく、モルヒズムの積も定義できるのである。積□× の第2成分には触れないので、idを□上の恒等式′とするモルヒズムのペア(L1_210E, id)を持ち上げることになります。では、ある応用例 から別の応用例 を因数分解するにはどうすればよいでしょうか。 $g' = g \circ (h \times \mathbf{id})$

ここで重要なのは、モルヒズムに対する積の作用である。普遍構文の3つ目は、普遍的に最も優れている対象を選ぶことである。このオブジェクトを□ ⇒ L1_1D44F と呼ぶことにしよう（これは一つのオブジェクトの記号的な名前と考え、Haskellの型クラス制約と混同しないように - 後で名前の付け方の違いについて説明します）。このオブジェクトには、独自のアプリケーション、つまり、(□ ⇒ □) × □ から □ へのモーフィズム（ここでは □ と呼びます）が付属しています。オブジェクト L1_1D44E ⇒ は、関数オブジェクトの他のどの候補も、その応用モルヒズム □ が □ を通して因数分解するように、それに一意にマッピングできる場合、最良のものです。このオブジェクトは我々のランキングによれば、他のどのオブジェクトよりも優れている。



The definition of the universal function object. This is the same diagram as above, but now the object $a \Rightarrow b$ is *universal*.

Formally:

A *function object* from a to b is an object $a \Rightarrow b$ together with the morphism

$$eval :: ((a \Rightarrow b) \times a) \to b$$

such that for any other object z with a morphism

$$g :: z \times a \to b$$

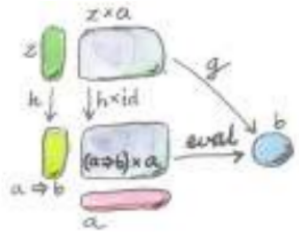
there is a unique morphism

$$h :: z \to (a \Rightarrow b)$$

that factors g through *eval*:

$$g = eval \circ (h \times \mathbf{id})$$

Of course, there is no guarantee that such an object $a \Rightarrow b$ exists for any pair of objects a and b in a given category. But it always does in **Set**. Moreover, in **Set**, this object is isomorphic to the hom-set $\mathbf{Set}(a, b)$.



普遍関数オブジェクトの定義です。これは上と同じ図ですが、今度はオブジェクト ☒ ⇒ ☒ が普遍的であることを示します。

Formally:

L1_1D44E から ☒ への関数オブジェクトはオブジェクト L1_1D44E ⇒ ☒ とモルヒズムとともに

$$eval :: ((a \Rightarrow b) \times a) \to b$$

を持つ他の任意のオブジェクト ☒ に対して、モルヒズム

$$g :: z \times a \to b$$

there is a unique morphism

$$h :: z \to (a \Rightarrow b)$$

that factors g through *eval*:

$$g = eval \circ (h \times \mathbf{id})$$

もちろん、このような目的語 L1_1D44E ⇒ ☒ が、あるカテゴリにおける任意の目的語 ☒ と ☒ の組に対して存在する保証はない。しかし、☒☒では必ず存在する。さらに、☒では、このオブジェクトはhom-set ☒(☒, ☒)に同型である。

This is why, in Haskell, we interpret the function type `a -> b` as the categorical function object $a \Rightarrow b$.

9.2 Currying

Let's have a second look at all the candidates for the function object. This time, however, let's think of the morphism g as a function of two variables, z and a.

$$g :: z \times a \to b$$

Being a morphism from a product comes as close as it gets to being a function of two variables. In particular, in **Set**, g is a function from pairs of values, one from the set z and one from the set a.

On the other hand, the universal property tells us that for each such g there is a unique morphism h that maps z to a function object $a \Rightarrow b$.

$$h :: z \to (a \Rightarrow b)$$

In **Set**, this just means that h is a function that takes one variable of type z and returns a function from a to b. That makes h a higher order function. Therefore the universal construction establishes a one-to-one correspondence between functions of two variables and functions of one variable returning functions. This correspondence is called *currying*, and h is called the curried version of g.

This correspondence is one-to-one, because given any g there is a unique h, and given any h you can always recreate the two-argument function g using the formula:

$$g = eval \circ (h \times \mathbf{id})$$

The function g can be called the *uncurried* version of h.

このことから、ハスケルでは関数型 `a -> b` をカテゴリ関数オブジェクト　L1_1D44E ⇒ ⊠ と解釈しているのです。

9.2 Currying

関数オブジェクトのすべての候補をもう一度見てみよう。ただし、今回はモルヒズム　　を ⊠ と ⊠ という2つの変数の関数として考えましょう。

$$g :: z \times a \to b$$

積からの形態素であることは、2変数の関数であることに限りなく近づきます。特に、⊠では、⊠は集合⊠からの値と集合⊠からの値の組からの関数である。一方、普遍特性はこのような　に対して、　を関数オブジェクト　L1_1D44E ⇒ ⊠に写す唯一の形態素　が存在することを教えてくれる。

$$h :: z \to (a \Rightarrow b)$$

⊠では、これは⊠が⊠型の変数を1つ取り、　L1_1D44E から　L1_1D44E への関数を返す関数であることを意味するだけです。そのため、　は高次関数となる。したがって、普遍構文は2変数の関数と関数を返す1変数の関数の間に1対1の対応を成立させる。この対応をカリー化と呼び、　L1_210E は　　のカリー化バージョンと呼ばれる。この対応は一対一で、任意の　　が与えられると一意な　L1_210E が存在し、任意の　L1_210E が与えられると必ず2引数関数　　を式を使って再作成できるからである。

$$g = eval \circ (h \times \mathbf{id})$$

関数　　は　　の非循環バージョンと呼ぶことができる。

Currying is essentially built into the syntax of Haskell. A function returning a function:

```
a -> (b -> c)
```

is often thought of as a function of two variables. That's how we read the un-parenthesized signature:

```
a -> b -> c
```

This interpretation is apparent in the way we define multi-argument functions. For instance:

```
catstr :: String -> String -> String
catstr s s' = s ++ s'
```

The same function can be written as a one-argument function returning a function — a lambda:

```
catstr' s = \s' -> s ++ s'
```

These two definitions are equivalent, and either can be partially applied to just one argument, producing a one-argument function, as in:

```
greet :: String -> String
greet = catstr "Hello "
```

Strictly speaking, a function of two variables is one that takes a pair (a product type):

Currying は基本的にHaskellの構文に組み込まれています。関数は関数を返します。

```
a -> (b -> c)
```

を返す関数は、しばしば2つの変数の関数とみなされます。そうやって、括弧のついていないシグネチャを読むのです。

```
a -> b -> c
```

この解釈は、複数引数の関数を定義する方法にも表れています。たとえば

```
catstr :: 文字列 -> 文字列 -> 文字列
catstr s s' = s ++ s'
```

同じ関数は、関数-ラムダを返す1引数関数として書くことができる。

```
catstr' s = \s' -> s ++ s'
```

この2つの定義は等価であり、どちらかを1つの引数に部分的に適用して、次のような1引数関数を生成することができます。

```
greet :: String -> String
greet = catstr "Hello "
```

厳密に言えば、2変数の関数はペア（積の型）を取る関数です。

```
(a, b) -> c
```

It's trivial to convert between the two representations, and the two (higher-order) functions that do it are called, unsurprisingly, `curry` and `uncurry`:

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f a b = f (a, b)
```

and

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (a, b) = f a b
```

Notice that `curry` is the *factorizer* for the universal construction of the function object. This is especially apparent if it's rewritten in this form:

```
factorizer :: ((a, b) -> c) -> (a -> (b -> c))
factorizer g = \a -> (\b -> g (a, b))
```

(As a reminder: A factorizer produces the factorizing function from a candidate.)

In non-functional languages, like C++, currying is possible but nontrivial. You can think of multi-argument functions in C++ as corresponding to Haskell functions taking tuples (although, to confuse things even more, in C++ you can define functions that take an explicit `std::tuple`, as well as variadic functions, and functions taking initializer lists).

You can partially apply a C++ function using the template `std::bind`. For instance, given a function of two strings:


```
(a, b) -> c
```

この2つの表現を変換するのは簡単で、これを行う2つの（高階）関数は、当然のことながら、`curry`と`uncurry`と呼ばれています。

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f a b = f (a, b)
```

and

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (a, b) = f a b
```

`curry`は関数オブジェクトの普遍的な構築のための因数分解であることに注意してください。これは特に、次のような形で書き直すとよくわかる。

```
factorizer :: ((a, b) -> c) -> (a -> (b -> c))
factorizer g = \a -> (\b -> g (a, b))
```

（備忘録として。因数分解器は、候補から因数分解関数を生成する）。C++のような非関数型言語では、`currying`は可能であるが、自明ではない。C++の多引数関数は、タプルを受け取るHaskellの関数に相当すると考えることができます（ただし、さらに混乱させるために、C++では、明示的に`std::tuple`を受け取る関数や、可変長関数、初期化リストを受け取る関数を定義することが可能です）。C++の関数は，テンプレート `std::bind` を使って，部分的に適用することができます．例えば，2つの文字列からなる関数が与えられたとします．

```
std::string catstr(std::string s1, std::string s2) {
    return s1 + s2;
}
```

you can define a function of one string:

```
using namespace std::placeholders;

auto greet = std::bind(catstr, "Hello ", _1);
std::cout << greet("Haskell Curry");
```

Scala, which is more functional than C++ or Java, falls somewhere in between. If you anticipate that the function you're defining will be partially applied, you define it with multiple argument lists:

```
def catstr(s1: String)(s2: String) = s1 + s2
```

Of course that requires some amount of foresight or prescience on the part of a library writer.

9.3 Exponentials

In mathematical literature, the function object, or the internal hom-object between two objects a and b, is often called the *exponential* and denoted by b^a. Notice that the argument type is in the exponent. This notation might seem strange at first, but it makes perfect sense if you think of the relationship between functions and products. We've already seen that we have to use the product in the universal construction of the internal hom-object, but the connection goes deeper than that.

This is best seen when you consider functions between finite types — types that have a finite number of values, like `Bool`, `Char`, or even `Int`


```
std::string catstr(std::string s1, std::string s2) {
    return s1 + s2;
}
```

という2つの文字列からなる関数があった場合，1つの文字列からなる関数を定義することができます．

```
using namespace std :: placeholders;

auto greet = std::bind(catstr, "Hello ", _1);
std::cout << greet("Haskell Curry");
```

C++やJavaよりも関数的なScalaは、その中間に位置します。定義する関数が部分的に適用されることが予想される場合、複数の引数リストで定義します。

```
def catstr(s1: String)(s2: String) = s1 + s2
```

もちろん、そのためには、ライブラリを書く側にある程度の先見性や予知性が必要ですが。

9.3 Exponentials

数学の文献では、関数オブジェクト、あるいは 2 つのオブジェクト L_1D44E と ⊠ の間の内部同値オブジェクトはしばしば指数と呼ばれ、 ⊠ と表記されます。引数の型が指数にあることに注意してください。この表記は最初は奇妙に思えるかもしれませんが、関数と積の関係を考えれば完璧に理解できます。内部のhom-objectの普遍的な構成で積を使わなければならないことはすでに見たが、その関連はもっと深い。これは、有限の型、つまりBoolやChar、あるいはIntのような有限の数の値を持つ型の間の関数を考えるとよくわかります。

or `Double`. Such functions, at least in principle, can be fully memoized or turned into data structures to be looked up. And this is the essence of the equivalence between functions, which are morphisms, and function types, which are objects.

For instance a (pure) function from `Bool` is completely specified by a pair of values: one corresponding to `False`, and one corresponding to `True`. The set of all possible functions from `Bool` to, say, `Int` is the set of all pairs of `Int`s. This is the same as the product `Int × Int` or, being a little creative with notation, $\texttt{Int}^2$.

For another example, let's look at the C++ type `char`, which contains 256 values (Haskell `Char` is larger, because Haskell uses Unicode). There are several functions in the part of the C++ Standard Library that are usually implemented using lookups. Functions like `isupper` or `isspace` are implemented using tables, which are equivalent to tuples of 256 Boolean values. A tuple is a product type, so we are dealing with products of 256 Booleans: `bool × bool × bool × ... × bool`. We know from arithmetics that an iterated product defines a power. If you "multiply" `bool` by itself 256 (or `char`) times, you get `bool` to the power of `char`, or $\texttt{bool}^{\texttt{char}}$.

How many values are there in the type defined as 256-tuples of `bool`? Exactly 2^{256}. This is also the number of different functions from `char` to `bool`, each function corresponding to a unique 256-tuple. You can similarly calculate that the number of functions from `bool` to `char` is 256^2, and so on. The exponential notation for function types makes perfect sense in these cases.

We probably wouldn't want to fully memoize a function from `int` or `double`. But the equivalence between functions and data types, if not always practical, is there. There are also infinite types, for instance lists, strings, or trees. Eager memoization of functions from those types

やDoubleのような有限の値を持つ型です。このような関数は、少なくとも原理的には、完全にメモ化したり、データ構造にして調べたりすることができます。そして、これが、モルヒズムである関数と、オブジェクトである関数型の間の等価性の本質なのです。例えば、Boolの（純粋な）関数は、Falseに対応する値とTrueに対応する値の組によって完全に規定される。BoolからIntへのすべての関数の集合は、Int sのすべてのペアの集合です。これは、積Int × Int、または表記を少し工夫してInt 2と同じです。別の例として、C++のcharという型を見てみましょう。C++標準ライブラリの一部には、通常ルックアップを使って実装される関数がいくつかあります。isupperやispaceのような関数は、256個のブール値のタプルに相当するテーブルを使用して実装されています。タプルは積の型なので、256個のブール値の積を扱っていることになる： bool × bool × bool × ... × x bool 。算数では、反復積がべき乗を定義することが知られている。もしboolにそれ自身を256回（またはchar）「掛ける」なら、boolのchar乗、またはbool charが得られます。bool の 256 タプルとして定義される型にはいくつの値があるのでしょうか？ちょうど2 256個です。これは、charからboolへの関数の数でもあり、それぞれの関数が一意の256タプルに対応します。同様に、boolからcharへの関数の個数は256 2個、などと計算できます。このような場合、関数型に対する指数表記は完全に意味をなします。intやdoubleからの関数を完全にメモすることは、おそらくないでしょう。しかし、関数とデータ型の間には、必ずしも実用的ではないにしても、等価性があるのです。また、リスト、文字列、木などの無限型もあります。これらの型からの関数を熱心にメモする

would require infinite storage. But Haskell is a lazy language, so the boundary between lazily evaluated (infinite) data structures and functions is fuzzy. This function vs. data duality explains the identification of Haskell's function type with the categorical exponential object — which corresponds more to our idea of *data*.

9.4 Cartesian Closed Categories

Although I will continue using the category of sets as a model for types and functions, it's worth mentioning that there is a larger family of categories that can be used for that purpose. These categories are called *Cartesian closed*, and **Set** is just one example of such a category.

A Cartesian closed category must contain:

1. The terminal object,
2. A product of any pair of objects, and
3. An exponential for any pair of objects.

If you consider an exponential as an iterated product (possibly infinitely many times), then you can think of a Cartesian closed category as one supporting products of an arbitrary arity. In particular, the terminal object can be thought of as a product of zero objects — or the zero-th power of an object.

What's interesting about Cartesian closed categories from the perspective of computer science is that they provide models for the simply typed lambda calculus, which forms the basis of all typed programming languages.

The terminal object and the product have their duals: the initial object and the coproduct. A Cartesian closed category that also supports

は無限のストレージを必要とします。しかし、Haskellは遅延言語なので、遅延評価される（無限の）データ構造と関数の境界は曖昧です。この関数とデータの二元性が、Haskellの関数型がカテゴリカルな指数オブジェクトと同一であることの説明となります - これは我々のデータに対する考え方に近いものです。

9.4 Cartesian Closed Categories

ここでは、型と関数のモデルとして集合のカテゴリを使い続けますが、この目的に使えるカテ ゴリにはもっと大きなファミリーがあることを述べておきます。これらのカテゴリはデカルト閉カテゴリと呼ばれ、□□はそのようなカテゴリの一例に過ぎない。デカルト閉カテゴリは以下を含まなければならない。

1. 終端オブジェクト、2.任意の対のオブジェクトの積、3.任意の対の指数関数。3. 任意の対のオブジェクトの指数。

指数関数が反復積(無限回)であるとするならば、デカルト閉カテゴリは任意のアリティの積をサポートするものであると考えることができる。特に、末端のオブジェクトは、ゼロのオブジェクトの積、あるいはオブジェクトのゼロ乗と考えることができる。コンピュータサイエンスの観点から見たデカルト閉カテゴリが興味深いのは、すべての型付きプログラミング言語の基礎となる単純型付けラムダ計算のモデルを提供している点である。終端オブジェクトと積は、その双対である初期オブジェクトと共積を持つ。をサポートするデカルト閉カテゴリもある。

those two, and in which product can be distributed over coproduct

$$a \times (b + c) = a \times b + a \times c$$
$$(b + c) \times a = b \times a + c \times a$$

is called a *bicartesian closed* category. We'll see in the next section that bicartesian closed categories, of which **Set** is a prime example, have some interesting properties.

9.5 Exponentials and Algebraic Data Types

The interpretation of function types as exponentials fits very well into the scheme of algebraic data types. It turns out that all the basic identities from high-school algebra relating numbers zero and one, sums, products, and exponentials hold pretty much unchanged in any bicartesian closed category theory for, respectively, initial and final objects, coproducts, products, and exponentials. We don't have the tools yet to prove them (such as adjunctions or the Yoneda lemma), but I'll list them here nevertheless as a source of valuable intuitions.

9.5.1 Zeroth Power

$$a^0 = 1$$

In the categorical interpretation, we replace 0 with the initial object, 1 with the final object, and equality with isomorphism. The exponential is the internal hom-object. This particular exponential represents the set of morphisms going from the initial object to an arbitrary object a. By the definition of the initial object, there is exactly one such morphism, so the hom-set $\mathbf{C}(0, a)$ is a singleton set. A singleton set is the terminal

をサポートし、積が共積の上に分布することができる直交閉カテゴリです。

$$a \times (b + c) = a \times b + a \times c$$
$$(b + c) \times a = b \times a + c \times a$$

をサポートするデカルト閉カテゴリをバイカルチエ閉カテゴリと呼ぶ。次節で、□ L1_1D41E がその代表例である二カルテス閉カテゴリが、いくつかの興味深い性質を 持っていることを見ることにします。

9.5 指数式と代数的データ型

指数関数としての関数型の解釈は、代数的データ型のスキームに非常によく合います。ゼロと1の数、和、積、指数に関する高校の代数学の基本的な恒等式は、それぞれ、初期および最終オブジェクト、共積、積、指数に関する二元的な閉カテゴリ理論でほとんど変わらないことが判明しています。それを証明する道具はまだないが(たとえば、adjunctions や Yoneda lemma)、それでも貴重な直観の源として、ここに列挙しておくことにする。

9.5.1 Zeroth Power

$$a^0 = 1$$

定型的解釈では、0を初期対象、1を最終対象、等式を同型に置き換える。指数関数が内部的なホムオブジェクトである。この特定の指数は、初期オブジェクトから任意のオブジェクト L1_1D44E に向かうモルヒズムの集合を表している。初期オブジェクトの定義により、そのようなモルヒズムはちょうど1つなので、ホム集合□(0, L1_1D44E) は単一集合である。シングルトン集合とは、末端

object in **Set**, so this identity trivially works in **Set**. What we are saying is that it works in any bicartesian closed category.

In Haskell, we replace 0 with `Void`; 1 with the unit type `()`; and the exponential with function type. The claim is that the set of functions from `Void` to any type `a` is equivalent to the unit type — which is a singleton. In other words, there is only one function `Void -> a`. We've seen this function before: it's called `absurd`.

This is a little bit tricky, for two reasons. One is that in Haskell we don't really have uninhabited types — every type contains the "result of a never ending calculation," or the bottom. The second reason is that all implementations of `absurd` are equivalent because, no matter what they do, nobody can ever execute them. There is no value that can be passed to `absurd`. (And if you manage to pass it a never ending calculation, it will never return!)

9.5.2 Powers of One

$$1^a = 1$$

This identity, when interpreted in **Set**, restates the definition of the terminal object: There is a unique morphism from any object to the terminal object. In general, the internal hom-object from a to the terminal object is isomorphic to the terminal object itself.

In Haskell, there is only one function from any type `a` to unit. We've seen this function before — it's called `unit`. You can also think of it as the function `const` partially applied to `()`.

9.5.3 First Power

$$a^1 = a$$

オブジェクトなので、この恒等式は　 では自明です。つまり、任意の二項閉カテゴリで動作するということである。Haskellでは、0をVoidに、1を単位型()に、そして指数を関数型に置き換えます。Voidから任意の型aへの関数の集合は、単位型と等価であり、それはシングルトンである、という主張である。つまり、Void -> a という関数は1つしかありません。 この関数は以前にも見たことがあります。この関数はちょっとやっかいです。1つは、Haskellでは実際に無人の型はなく、すべての型が「終わらない計算の結果」、つまり底辺を含んでいるからです。もう一つは、不条理な実装はすべて等価であり、何をしようが誰も実行できないからである。不条理に渡すことのできる値はありません。（そして、なんとか終わらない計算を渡したとしても、決して戻ってきません！）

9.5.2 Powers of One

$$1^a = 1$$

この恒等式を　で解釈すると、終端オブジェクトの定義が再表現される。任意のオブジェクトから終端オブジェクトへの一意なモーフィズムが存在する。一般に、 Ll_lD44E からターミナルオブジェクトへの内部ホムオブジェクトはターミナルオブジェクト自体と同型である。Haskellでは、任意の型aからunitへの関数が1つだけ存在する。この関数は以前にも見たことがあり、unit と呼ばれています。また、この関数は const を部分的に () に適用したものと考えることもできます。

9.5.3 First Power

$$a^1 = a$$

This is a restatement of the observation that morphisms from the terminal object can be used to pick "elements" of the object a. The set of such morphisms is isomorphic to the object itself. In **Set**, and in Haskell, the isomorphism is between elements of the set a and functions that pick those elements, `() -> a`.

9.5.4 Exponentials of Sums

$$a^{b+c} = a^b \times a^c$$

Categorically, this says that the exponential from a coproduct of two objects is isomorphic to a product of two exponentials. In Haskell, this algebraic identity has a very practical, interpretation. It tells us that a function from a sum of two types is equivalent to a pair of functions from individual types. This is just the case analysis that we use when defining functions on sums. Instead of writing one function definition with a `case` statement, we usually split it into two (or more) functions dealing with each type constructor separately. For instance, take a function from the sum type `(Either Int Double)`:

```
f :: Either Int Double -> String
```

It may be defined as a pair of functions from, respectively, `Int` and `Double`:

```
f (Left n) = if n < 0 then "Negative int" else "Positive int"
f (Right x) = if x < 0.0 then "Negative double" else "Positive double"
```

Here, `n` is an `Int` and `x` is a `Double`.

これは、終端オブジェクトからの形態素はオブジェクトaの「要素」を選ぶために使われる という観測の再提示です。 このような形態素の集合はオブジェクト自体に同型です。 ☒とHaskellでは、同型性は集合aの要素とそれらの要素を選ぶ関数、 () -> aの間にあります。

9.5.4 Exponentials of Sums

$$a^{b+c} = a^b \times a^c$$

カテゴリ的には、これは2つのオブジェクトの共積から得られる指数が2つの指数の積と同型であることを意味します。Haskellでは、この代数的恒等式は非常に実用的な解釈をする。これは、2つの型の和から得られる関数は、個々の型から得られる関数のペアと等価であることを教えてくれる。これは、和に対する関数を定義するときに使う格解析と同じです。通常、1つの関数定義にcase文を記述するのではなく、各型の構成要素を個別に扱う2つ（以上）の関数に分割して記述します。例えば、sum型(Either Int Double)の関数を例にとりますと
.

```
f :: Either Int Double -> String
```

これは，それぞれInt型とDouble型からなる関数のペアとして定義することができます.
Double:

```
f ( 左 n) = if n < 0 then "Negative int" else "Positive int".
f ( 右 x) = if x < 0.0 then "Negative double" else "Positive double".
```

ここで、nはInt、xはDoubleです。

9.5.5 Exponentials of Exponentials

$$(a^b)^c = a^{b \times c}$$

This is just a way of expressing currying purely in terms of exponential objects. A function returning a function is equivalent to a function from a product (a two-argument function).

9.5.6 Exponentials over Products

$$(a \times b)^c = a^c \times b^c$$

In Haskell: A function returning a pair is equivalent to a pair of functions, each producing one element of the pair.

It's pretty incredible how those simple high-school algebraic identities can be lifted to category theory and have practical application in functional programming.

9.6 Curry-Howard Isomorphism

I have already mentioned the correspondence between logic and algebraic data types. The `Void` type and the unit type `()` correspond to false and true. Product types and sum types correspond to logical conjunction ∧ (AND) and disjunction ∨ (OR). In this scheme, the function type we have just defined corresponds to logical implication ⇒. In other words, the type `a -> b` can be read as "if a then b."

According to the Curry-Howard isomorphism, every type can be interpreted as a proposition — a statement or a judgment that may be true or false. Such a proposition is considered true if the type is inhabited and false if it isn't. In particular, a logical implication is true if the function type corresponding to it is inhabited, which means that there

9.5.5 Exponentials of Exponentials

$$(a^b)^c = a^{b \times c}$$

これは純粋に指数オブジェクトでキュアリングを表現しているに過ぎません。関数を返す関数は、積からの関数（2引数関数）と等価です。

9.5.6 Exponentials over Products

$$(a \times b)^c = a^c \times b^c$$

Haskellでは ペアを返す関数は、それぞれがペアの1つの要素を生成する関数のペアと等価です。高校で習うような簡単な代数的恒等式が、カテゴリ理論に持ち込まれて、関数型プログラミングに実用化されるのは、かなりすごいことです。

9.6 Curry-Howard Isomorphism

論理型と代数型の対応関係についてはすでに述べた。Void型と単位型()はfalseとtrueに対応する。積型と和型は論理積∧(AND)と論理和∨(OR)に対応する。この方式では、今定義した関数型が論理的含意⇒に相当する。つまり、a→b型は、"if a then b "と読める。カレー・ハワード同型によれば、すべての型は命題、つまり真か偽かを問わない文や判断として解釈される。このような命題は、その型が宿っていれば真、そうでなければ偽とみなされる。特に、論理的含意はそれに対応する関数型が居住していれば真であり、それは以下 のことを意味する。

exists a function of that type. An implementation of a function is therefore a proof of a theorem. Writing programs is equivalent to proving theorems. Let's see a few examples.

Let's take the function `eval` we have introduced in the definition of the function object. Its signature is:

```
eval :: ((a -> b), a) -> b
```

It takes a pair consisting of a function and its argument and produces a result of the appropriate type. It's the Haskell implementation of the morphism:

$$eval :: (a \Rightarrow b) \times a \to b$$

which defines the function type $a \Rightarrow b$ (or the exponential object b^a). Let's translate this signature to a logical predicate using the Curry-Howard isomorphism:

$$((a \Rightarrow b) \wedge a) \Rightarrow b$$

Here's how you can read this statement: If it's true that b follows from a, and a is true, then b must be true. This makes perfect intuitive sense and has been known since antiquity as *modus ponens*. We can prove this theorem by implementing the function:

```
eval :: ((a -> b), a) -> b
eval (f, x) = f x
```

If you give me a pair consisting of a function `f` taking `a` and returning `b`, and a concrete value `x` of type `a`, I can produce a concrete value of type

はその型の関数が存在することを意味します。したがって、ある関数の実装は定理の証明になります。プログラムを書くことは、定理を証明することに等しいのです。少し例を見てみましょう。関数オブジェクトの定義で紹介した関数 `eval` を例にとってみましょう。そのシグネチャは

```
eval :: ((a -> b), a) -> b
```

関数とその引数のペアを受け取り、適切な型の結果を生成します。これはモルヒズムのHaskell実装です。

$$eval :: (a \Rightarrow b) \times a \to b$$

これは、関数型 ☒ ⇒ ☒（または指数オブジェクト ☒）を定義するモルヒズムのHaskell実装です。この署名をCurryHowardの同型性を使って論理的な述語に変換してみましょう。

$$((a \Rightarrow b) \wedge a) \Rightarrow b$$

この文はこう読むことができる。☒が　Ll_1D44E から導かれるのが真で、　Ll_1D44E が真なら、☒は真でなければならない。これは直観的に完全に理解でき、古くからmodus ponensとして知られている。この定理は関数を実装することで証明できる。

```
eval :: ((a -> b), a) -> b
eval (f, x) = f x
```

a をとり b を返す関数 f と a 型の具体的な値 x からなるペアを与えると、私は a 型の具体的な値を生成することができます。

`b` by simply applying the function `f` to `x`. By implementing this function I have just shown that the type `((a -> b), a) -> b` is inhabited. Therefore *modus ponens* is true in our logic.

How about a predicate that is blatantly false? For instance: if a or b is true then a must be true.

$$a \vee b \Rightarrow a$$

This is obviously wrong because you can choose an a that is false and a b that is true, and that's a counter-example.

Mapping this predicate into a function signature using the Curry-Howard isomorphism, we get:

```
Either a b -> a
```

Try as you may, you can't implement this function — you can't produce a value of type `a` if you are called with the `Right` value. (Remember, we are talking about *pure* functions.)

Finally, we come to the meaning of the `absurd` function:

```
absurd :: Void -> a
```

Considering that `Void` translates into false, we get:

$$false \Rightarrow a$$

Anything follows from falsehood (*ex falso quodlibet*). Here's one possible proof (implementation) of this statement (function) in Haskell:

```
absurd (Void a) = absurd a
```

where `Void` is defined as:

この関数を実装することによって、私は型`((a -> b), a) -> b`が存在することを示したことになる。したがって、我々の論理では `modus ponens` が真となる。では、あからさまに間違っている述語はどうでしょうか？例えば：もし□または　`Ll_lD44E` が真ならば、　`Ll_lD44E` は真でなければならない。

$$a \vee b \Rightarrow a$$

これは明らかに間違っています。なぜなら、偽である□と真である□を選ぶことができるからで、これは反例となります。この述語をCurryHowardの同型性を使って関数シグネチャにマッピングすると、次のようになります。

```
Either a b -> a
```

試してみたが、この関数は実装できない - 右の値で呼ばれたらa型の値を生成できない。(純粋な関数について話していることを忘れないでください。) 最後に、不条理な関数の意味について説明します。

```
absurd :: Void -> a
```

Voidがfalseに変換されることを考慮すると、次のようになります。

$$false \Rightarrow a$$

何でも偽りから導かれる(ex falso quodlibet)。この文(関数)をHaskellで証明(実装)する可能性のあるものを一つ挙げてみよう。

```
absurd (Void a) = absurd a
```

where `Void` is defined as:

```
newtype Void = Void Void
```

As always, the type `Void` is tricky. This definition makes it impossible to construct a value because in order to construct one, you would need to provide one. Therefore, the function `absurd` can never be called.

These are all interesting examples, but is there a practical side to Curry-Howard isomorphism? Probably not in everyday programming. But there are programming languages like Agda or Coq, which take advantage of the Curry-Howard isomorphism to prove theorems.

Computers are not only helping mathematicians do their work — they are revolutionizing the very foundations of mathematics. The latest hot research topic in that area is called Homotopy Type Theory, and is an outgrowth of type theory. It's full of Booleans, integers, products and coproducts, function types, and so on. And, as if to dispel any doubts, the theory is being formulated in Coq and Agda. Computers are revolutionizing the world in more than one way.

9.7 Bibliography

1. Ralph Hinze, Daniel W. H. James, Reason Isomorphically![1]. This paper contains proofs of all those high-school algebraic identities in category theory that I mentioned in this chapter.

[1] http://www.cs.ox.ac.uk/ralf.hinze/publications/WGP10.pdf


```
newtype Void = Void Void
```


例によって、Voidという型はやっかいだ。この定義では、値を構成するためには、値を提供する必要があるため、値を構成することは不可能です。したがって、関数 absurd は決して呼び出すことができません。これらはすべて興味深い例ですが、Curry-Howard同型性には実用的な側面があるのでしょうか？おそらく、日常のプログラミングではありえないだろう。しかし、AgdaやCoqのように、カレー・ハワード同型性を利用して定理を証明するプログラミング言語がある。コンピュータは、数学者の仕事を助けるだけでなく、数学の基礎そのものに革命を起こしているのです。その分野での最新のホットな研究テーマは「ホモトピー型理論」と呼ばれ、型理論の発展型である。ブール語、整数、積と共積、関数型などなど、盛りだくさんだ。そして、その疑念を払拭するかのように、この理論はCoqやAgdaで定式化されつつある。コンピュータが世界に革命を起こしているのは、1つだけではありません。

9.7 Bibliography

1. Ralph Hinze, Daniel W. H. James, Reason Isomorphically！1．この論文には、この章で紹介した、カテゴリ理論における高校生の代数的恒等式の証明がすべて含まれている。

[1] http://www.cs.ox.ac.uk/ralf.hinze/publications/WGP10.pdf