3

Categories Great and Small

You can get real appreciation for categories by studying a variety of examples. Categories come in all shapes and sizes and often pop up in unexpected places. We'll start with something really simple.

3.1 No Objects

The most trivial category is one with zero objects and, consequently, zero morphisms. It's a very sad category by itself, but it may be important in the context of other categories, for instance, in the category of all categories (yes, there is one). If you think that an empty set makes sense, then why not an empty category?

3

Categories Great and Small

カテゴリーを理解するためには、様々な事例を研究することが必要です。カテゴリーには様々な形や大きさがあり、思いがけないところに現れることもよくあります。まずは簡単なものから始めてみましょう。

3.1 No Objects

最もつまらないカテゴリーとは、対象がゼロで、その結果、形態素もゼロのカテゴリーである。それ自身はとても悲しいカテゴリですが，他のカテゴリ，たとえば，すべてのカテゴリのカテゴリ（そうです，1つあります）の文脈では重要かもしれません．もし、空の集合が意味をなすと考えるなら、空のカテゴリーでもいいのでは ないか？

3.2 Simple Graphs

You can build categories just by connecting objects with arrows. You can imagine starting with any directed graph and making it into a category by simply adding more arrows. First, add an identity arrow at each node. Then, for any two arrows such that the end of one coincides with the beginning of the other (in other words, any two *composable* arrows), add a new arrow to serve as their composition. Every time you add a new arrow, you have to also consider its composition with any other arrow (except for the identity arrows) and itself. You usually end up with infinitely many arrows, but that's okay.

Another way of looking at this process is that you're creating a category, which has an object for every node in the graph, and all possible *chains* of composable graph edges as morphisms. (You may even consider identity morphisms as special cases of chains of length zero.)

Such a category is called a *free category* generated by a given graph. It's an example of a free construction, a process of completing a given structure by extending it with a minimum number of items to satisfy its laws (here, the laws of a category). We'll see more examples of it in the future.

3.3 Orders

And now for something completely different! A category where a morphism is a relation between objects: the relation of being less than or equal. Let's check if it indeed is a category. Do we have identity morphisms? Every object is less than or equal to itself: check! Do we have composition? If $a \leqslant b$ and $b \leqslant c$ then $a \leqslant c$: check! Is composition as-

3.2 Simple Graphs

オブジェクトを矢印で結ぶだけで、カテゴリーを構築することができる。任意の有向グラフから始めて、矢印を増やすだけでカテゴリーにすることが想像できる。まず、各ノードに同一性の矢印を追加する。次に、一方の終点と他方の始点が一致するような任意の2つの矢印（言い換えれば、任意の2つの合成可能な矢印）に対して、それらの合成となる新しい矢印を追加する。新しい矢印を追加するたびに、他の矢印（同一性のある矢印を除く）およびそれ自身との合成も考慮しなければならない。通常、矢印は無限に増えていくが、それでも構わない。別の見方をすれば、グラフの各ノードに対してオブジェクトを持ち、合成可能なグラフのエッジの連鎖をすべて形態素として持つカテゴリーを作っていることになる。（長さ0の鎖の特殊な場合として，同一性モルヒズムを考えてもよい）．このようなカテゴリを、与えられたグラフが生成する自由カテゴリと呼ぶ。これは自由な構築の一例であり，与えられた構造を，その法則（ここではカテゴリの法則）を満たす最小限の項目で拡張することによって完成させるプロセスである．今後、もっと多くの例を見ていくことにしよう。

3.3 Orders

そして、今度はまったく別のものです モルヒズムがオブジェクト間の関係であるカテゴリー：以下であるか等しいかの関係である。これが本当にカテゴリーなのかどうか調べてみましょう．同一性モルヒズムはあるのだろうか？すべてのオブジェクトはそれ自身と等しいかそれ以下である．合成はあるか？ ☒ ☒ ☒ ならば ☒ ☒ ☒：確認！ 合成は

sociative? Check! A set with a relation like this is called a *preorder*, so a preorder is indeed a category.

You can also have a stronger relation, that satisfies an additional condition that, if $a \leqslant b$ and $b \leqslant a$ then a must be the same as b. That's called a *partial order*.

Finally, you can impose the condition that any two objects are in a relation with each other, one way or another; and that gives you a *linear order* or *total order*.

Let's characterize these ordered sets as categories. A preorder is a category where there is at most one morphism going from any object a to any object b. Another name for such a category is "thin." A preorder is a thin category.

A set of morphisms from object a to object b in a category $\mathbf{C}$ is called a *hom-set* and is written as $\mathbf{C}(a, b)$ (or, sometimes, $\mathbf{Hom}_{\mathbf{C}}(a, b)$). So every hom-set in a preorder is either empty or a singleton. That includes the hom-set $\mathbf{C}(a, a)$, the set of morphisms from a to a, which must be a singleton, containing only the identity, in any preorder. You may, however, have cycles in a preorder. Cycles are forbidden in a partial order.

It's very important to be able to recognize preorders, partial orders, and total orders because of sorting. Sorting algorithms, such as quicksort, bubble sort, merge sort, etc., can only work correctly on total orders. Partial orders can be sorted using topological sort.

3.4 Monoid as Set

Monoid is an embarrassingly simple but amazingly powerful concept. It's the concept behind basic arithmetics: Both addition and multiplication form a monoid. Monoids are ubiquitous in programming. They

社会的か？チェック！ このような関係を持つ集合は前置関係と呼ばれ、前置関係は確かにカテゴリである。もっと強い関係もあります。それは、 ならば は と同じでなければならないという追加の条件を満たすものです。これは部分順序と呼ばれます。最後に、2つの対象が互いに何らかの関係にあることを条件とすれば、線形順序や全順序が得られる。これらの順序付き集合をカテゴリーとして特徴づけよう。前置順位とは、任意のオブジェクト から任意のオブジェクト に向かうモルヒズムがせいぜい1つしかないようなカテゴリである。このようなカテゴリの別の名前は ”薄い” である。前置詞は薄いカテゴリである。カテゴリ におけるオブジェクト L1_1D44E からオブジェクト へのモルヒズムのセットはホムセットと呼ばれ、 (or sometimes, Hom , L1_1D44E) と表記される。したがって、前置詞のホムセットはすべて空かシングルトンである。 L1_1D44E から L1_1D44E への形態素の集合であるホム集合 (L1_1D44E) は、どんな前置詞でも単項式でなければならず、恒等式を含んでいます。しかし、前置順序ではサイクルを持つことができる。サイクルは部分順序では禁止されています。前置順、部分順、全体順を認識できることは、並べ替えのために非常に重要である。クイックソート、バブルソート、マージソートなどのソートアルゴリズムは、全順序に対してのみ正しく働くことができます。部分オーダーはトポロジカルソートを使ってソートすることができます。

3.4 Monoid as Set

Monoid は恥ずかしいほど単純ですが、驚くほど強力な概念です。これは基本的な算術の背後にある概念です。足し算と掛け算はともにモノイドを形成します。モノイドはプログラミングの世界ではどこにでもあるものです。これらは

show up as strings, lists, foldable data structures, futures in concurrent programming, events in functional reactive programming, and so on.

Traditionally, a monoid is defined as a set with a binary operation. All that's required from this operation is that it's associative, and that there is one special element that behaves like a unit with respect to it.

For instance, natural numbers with zero form a monoid under addition. Associativity means that:

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

(In other words, we can skip parentheses when adding numbers.)

The neutral element is zero, because:

$$0 + a = a$$

and

$$a + 0 = a$$

The second equation is redundant, because addition is commutative ($a + b = b + a$), but commutativity is not part of the definition of a monoid. For instance, string concatenation is not commutative and yet it forms a monoid. The neutral element for string concatenation, by the way, is an empty string, which can be attached to either side of a string without changing it.

In Haskell we can define a type class for monoids — a type for which there is a neutral element called `mempty` and a binary operation called `mappend`:

```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m
```


文字列、リスト、折りたたみ可能なデータ構造、並行処理プログラミングにおけるフューチャー、関数型反応プログラミングにおけるイベントなどです。伝統的に、モノイドは二項演算を持つ集合として定義されます。この演算に必要なのは、演算が連想的であることと、演算に対して単位のように振る舞う特別な要素が1つあることです。例えば、0を含む自然数は加算のもとでモノイドを形成します。連想性とは、次のようなことです。

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

（ということです（つまり、数を足すときに括弧を省略できる）。中立的な要素はゼロですから。

$$0 + a = a$$

and

$$a + 0 = a$$

足し算は可換なので、2番目の式は冗長ですが（□ = □）、可換性はモノイドの定義には含まれません。例えば、文字列の連結は可換ではないのにモノイドを形成する。ちなみに文字列連結の中性要素は空文字列で、これは文字列のどちら側に付けても変化しない。Haskellではモノイドのための型クラス、つまりmemptyという中立的な要素とmappendという二項演算が存在する型を定義することができます.

```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m
```

The type signature for a two-argument function, `m -> m -> m`, might look strange at first, but it will make perfect sense after we talk about currying. You may interpret a signature with multiple arrows in two basic ways: as a function of multiple arguments, with the rightmost type being the return type; or as a function of one argument (the leftmost one), returning a function. The latter interpretation may be emphasized by adding parentheses (which are redundant, because the arrow is right-associative), as in: `m -> (m -> m)`. We'll come back to this interpretation in a moment.

Notice that, in Haskell, there is no way to express the monoidal properties of `mempty` and `mappend` (i.e., the fact that `mempty` is neutral and that `mappend` is associative). It's the responsibility of the programmer to make sure they are satisfied.

Haskell classes are not as intrusive as C++ classes. When you're defining a new type, you don't have to specify its class up front. You are free to procrastinate and declare a given type to be an instance of some class much later. As an example, let's declare `String` to be a monoid by providing the implementation of `mempty` and `mappend` (this is, in fact, done for you in the standard Prelude):

```
instance Monoid String where
    mempty = ""
    mappend = (++)
```

Here, we have reused the list concatenation operator `(++)`, because a `String` is just a list of characters.

A word about Haskell syntax: Any infix operator can be turned into a two-argument function by surrounding it with parentheses. Given two strings, you can concatenate them by inserting ++ between them:

2引数の関数に対する型シグネチャ、`m -> m -> m` は、最初は奇妙に見えるかもしれないが、 currying の話をすれば完璧に理解できるだろう。複数の矢印を持つシグネチャは2つの基本的な方法で解釈できます：複数の引数の関数で、一番右の型が戻り値の型である、または、1つの引数の関数（一番左の型）で、関数を返す、です。後者の場合、括弧をつけることで強調することができる（括弧は右結合なので冗長である）： `m -> (m -> m)`．この解釈についてはまた後ほど説明します．Haskellでは、memptyとmappendのモノイダル特性（memptyが中立でmappendが連想的であるという事実）を表現する方法はないことに注意してください。それらが満たされるようにするのは、プログラマーの責任です。Haskellのクラスは、C++のクラスのように押しつけがましくありません。新しい型を定義するときに、前もってそのクラスを指定する必要はない。新しい型を定義するときに、前もってそのクラスを指定する必要はありません。与えられた型があるクラスのインスタンスであることを宣言するのは、ずっと先延ばしにしてもかまいません。例として、`mempty` と `mappend` の実装を用意して `String` がモノイドであることを宣言してみましょう（実際、これは標準の `Prelude` で行われています）。

```
instance Monoid String where
    mempty = ""
    mappend = (++)
```

ここで、Stringは単なる文字のリストなので、リスト連結演算子(++)を再利用しています。Haskellの構文について一言。Haskellの構文について一言。どんなinfix演算子でも、括弧で囲むことで2引数の関数にすることができます。2つの文字列があるとき、その間に++を挿入すると、2つの文字列を連結することができます。

```
"Hello " ++ "world!"
```

or by passing them as two arguments to the parenthesized `(++)`:

```
(++) "Hello " "world!"
```

Notice that arguments to a function are not separated by commas or surrounded by parentheses. (This is probably the hardest thing to get used to when learning Haskell.)

It's worth emphasizing that Haskell lets you express equality of functions, as in:

```
mappend = (++)
```

Conceptually, this is different than expressing the equality of values produced by functions, as in:

```
mappend s1 s2 = (++) s1 s2
```

The former translates into equality of morphisms in the category **Hask** (or **Set**, if we ignore bottoms, which is the name for never-ending calculations). Such equations are not only more succinct, but can often be generalized to other categories. The latter is called *extensional* equality, and states the fact that for any two input strings, the outputs of `mappend` and `(++)` are the same. Since the values of arguments are sometimes called *points* (as in: the value of f at point x), this is called pointwise equality. Function equality without specifying the arguments is described as *point-free*. (Incidentally, point-free equations often involve composition of functions, which is symbolized by a point, so this might be a little confusing to the beginner.)

The closest one can get to declaring a monoid in C++ would be to use the (proposed) syntax for concepts.


```
"Hello " ++ "world!"
```


あるいは、括弧で囲んだ(++)に2つの引数を渡せば、それらを連結することができます。


```
(++) "Hello " "world!"
```


関数への引数は、カンマで区切られたり、括弧で囲まれたりしていないことに注意してください。（これがHaskellを学ぶ上で一番慣れないことかもしれません。）Haskellでは、次のように関数の等式を表現できることを強調しておきます。


```
mappend = (++)
```


概念的には、これは関数が生成する値の等価性を表現するのとは異なります。


```
mappend s1 s2 = (++) s1 s2
```


前者はカテゴリーHask（または□□、これは終わりのない計算の名前です）におけるモルヒズムの等式に変換されます。このような方程式はより簡潔であるだけでなく、しばしば他のカテゴリに一般化することができる。後者は外延的等式と呼ばれ、任意の2つの入力文字列に対して、mappendと(++)の出力が同じであることを述べている。引数の値を点と呼ぶことがあるので（点　での　の値というように）、これを点等式と呼ぶ。引数を指定しない関数の等式を無点等式と表現する。（ちなみに無点方程式は関数の合成を伴うことが多く、これは点で象徴されるので、初心者は少し混乱するかもしれません）。C++でモノイドを宣言するのに一番近いのは、概念のための(提案されている)構文を使うことでしょう。

```
template<class T>
  T mempty = delete;

template<class T>
  T mappend(T, T) = delete;

template<class M>
  concept bool Monoid = requires (M m) {
    { mempty<M> } -> M;
    { mappend(m, m); } -> M;
  };
```

The first definition uses a value template (also proposed). A polymorphic value is a family of values — a different value for every type.

The keyword `delete` means that there is no default value defined: It will have to be specified on a case-by-case basis. Similarly, there is no default for `mappend`.

The concept `Monoid` is a predicate (hence the `bool` type) that tests whether there exist appropriate definitions of `mempty` and `mappend` for a given type `M`.

An instantiation of the Monoid concept can be accomplished by providing appropriate specializations and overloads:

```
template<>
std::string mempty<std::string> = {""};

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



```
template<class T>
  T mempty = delete;

template<class T>
  T mappend(T, T) = delete;

template<class M>
  概念 bool Monoid = requires (M m) {.
    { mempty<M> } -> M;
    { mappend(m, m); } -> M;
  };
```

最初の定義では、（これも提案されている）値テンプレートを使っています。多相値は値のファミリーで、すべての型に対して異なる値を持つ。deleteというキーワードは、デフォルト値が定義されていないことを意味する。ケースバイケースで指定する必要があります。同様に、mappendにもデフォルトはありません。Monoidという概念は、与えられた型Mに対してmemptyとmappendの適切な定義が存在するかどうかをテストする述語（そのためbool型になっています）です。Monoid概念のインスタンス化は、適切な特殊化とオーバーロードを提供することで 実現できます。

```
template<>
std::string mempty<std::string> = {""};

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

3.5 Monoid as Category

That was the “familiar” definition of the monoid in terms of elements of a set. But as you know, in category theory we try to get away from sets and their elements, and instead talk about objects and morphisms. So let’s change our perspective a bit and think of the application of the binary operator as “moving” or “shifting” things around the set.

For instance, there is the operation of adding 5 to every natural number. It maps 0 to 5, 1 to 6, 2 to 7, and so on. That’s a function defined on the set of natural numbers. That’s good: we have a function and a set. In general, for any number n there is a function of adding n — the “adder” of n.

How do adders compose? The composition of the function that adds 5 with the function that adds 7 is a function that adds 12. So the composition of adders can be made equivalent to the rules of addition. That’s good too: we can replace addition with function composition.

But wait, there’s more: There is also the adder for the neutral element, zero. Adding zero doesn’t move things around, so it’s the identity function in the set of natural numbers.

Instead of giving you the traditional rules of addition, I could as well give you the rules of composing adders, without any loss of information. Notice that the composition of adders is associative, because the composition of functions is associative; and we have the zero adder corresponding to the identity function.

An astute reader might have noticed that the mapping from integers to adders follows from the second interpretation of the type signature of `mappend` as `m -> (m -> m)`. It tells us that `mappend` maps an element of a monoid set to a function acting on that set.

3.5 Monoid as Category

これが、集合の要素という観点からのモノイドの「お馴染みの」定義でした。しかし、ご存知のように、カテゴリ理論では、集合とその要素から離れ、代わりにオブジェクトとモルヒズムについて話そうとします。そこで、少し視点を変えて、二項演算子の適用を、集合の周りのものを「動かす」「ずらす」ことと考えてみましょう。たとえば、すべての自然数に5を足す演算があります。0を5に、1を6に、2を7に、といった具合に対応させるのです。これは、自然数の集合に対して定義された関数である。関数と集合が揃ったのだからいいじゃないか。一般に、どんな数nに対しても、⍰を足す関数、つまり⍰の「加算器」が存在する。加算器はどのように合成されるのでしょうか？5を足す関数と7を足す関数の合成は、12を足す関数になります。つまり、加算器の合成は足し算のルールと等価にすることができるのです。足し算を関数の合成に置き換えることができるのです。しかし、待てよ、まだある。中立的な要素である「0」の加算器もある。ゼロを加えても物事は動かないので、自然数の集合では恒等関数となる。従来の足し算のルールを説明する代わりに、足し算の合成のルールを説明しても、情報の損失はありませんね。関数の合成は連想的なので、加算器の合成は連想的であることに注意してください。そして、恒等関数に対応するゼロ加算器を持っています。鋭い読者は，整数から加算器への写像が，`mappend`の型署名の2番目の解釈である `m -> (m -> m)` に従っていることに気づいたかもしれない．これは，`mappend` がモノイド集合の要素をその集合に作用する関数に写像することを表しています．

Now I want you to forget that you are dealing with the set of natural numbers and just think of it as a single object, a blob with a bunch of morphisms — the adders. A monoid is a single object category. In fact the name monoid comes from Greek *mono*, which means single. Every monoid can be described as a single object category with a set of morphisms that follow appropriate rules of composition.

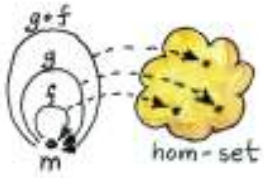
String concatenation is an interesting case, because we have a choice of defining right appenders and left appenders (or *prependers*, if you will). The composition tables of the two models are a mirror reverse of each other. You can easily convince yourself that appending “bar” after “foo” corresponds to prepending “foo” after prepending “bar”.

You might ask the question whether every categorical monoid — a one-object category — defines a unique set-with-binary-operator monoid. It turns out that we can always extract a set from a single-object category. This set is the set of morphisms — the adders in our example. In other words, we have the hom-set $\mathbf{M}(m, m)$ of the single object m in the category $\mathbf{M}$. We can easily define a binary operator in this set: The monoidal product of two set-elements is the element corresponding to

ここで、自然数の集合を扱っていることを忘れて、それを1つのオブジェクト、つまり、たくさんの形態素（加算器）を持つ塊として考えてみてください。モノイドは単一オブジェクトのカテゴリです。実際、モノイドという名前はギリシャ語のmonoからきていて、単一という意味です。すべてのモノイドは、適切な合成規則に従ったモルヒズムのセットを持つ単一のオブジェクトのカテゴリとして記述することができます。

文字列の連結は、右アペンダーと左アペンダー（またはプリペンダー）を定義する 選択肢があるので、興味深いケースです。この2つのモデルの合成表は、互いに鏡像反転している。foo ”の後に ”bar ”を付加することは、”bar ”の前に ”foo ”を付加することに対応することは容易に納得がいくだろう。すべてのカテゴリカル・モノイド（1オブジェクト・カテゴリ）は、一意なセット・ウィズ・バイナリー・オペレーター・モノイドを定義するのか、という疑問があるかもしれない。しかし、単一目的格から必ず集合を抽出できることがわかった。この集合は、モルヒズム（この例では、加算器）の集合である。つまり、カテゴリ の単一オブジェクト Ll_1D45A のホム集合 □(□) がある。この集合では、二項演算子を簡単に定義できる。二つの集合要素のモノイダル積は、以下の要素に対応するものである。

the composition of the corresponding morphisms. If you give me two elements of $\mathbf{M}(m,m)$ corresponding to f and g, their product will correspond to the composition $f \circ g$. The composition always exists, because the source and the target for these morphisms are the same object. And it's associative by the rules of category. The identity morphism is the neutral element of this product. So we can always recover a set monoid from a category monoid. For all intents and purposes they are one and the same.

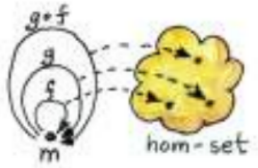


Monoid hom-set seen as morphisms and as points in a set.

There is just one little nit for mathematicians to pick: morphisms don't have to form a set. In the world of categories there are things larger than sets. A category in which morphisms between any two objects form a set is called locally small. As promised, I will be mostly ignoring such subtleties, but I thought I should mention them for the record.

A lot of interesting phenomena in category theory have their root in the fact that elements of a hom-set can be seen both as morphisms, which follow the rules of composition, and as points in a set. Here, composition of morphisms in $\mathbf{M}$ translates into monoidal product in the set $\mathbf{M}(m,m)$.

に対応する要素であり、対応するモルヒズムの合成である。 （ L1_1D45A）の2つの要素を□と□に対応させると、それらの積は合成 □ □ に対応することになる。これらのモルヒズムのソースとターゲットが同じオブジェクトであるため、合成は常に存在する。そして、カテゴリの規則によって連想的である。同一性モルヒズムはこの積の中性要素である。だから、集合モノイドは常にカテゴリモノイドから復元できるのです。どこから見ても、それらは同じものなのです。



モルヒズムとして、また集合の中の点として見られるモノイドホムセット。

数学者が選ぶべきは、1つだけ小さな些細なことだ： 形態素は集合を形成する必要はないのだ。カテゴリーの世界では、集合よりも大きなものが存在する。任意の2つの対象の間の形態素が集合を形成するようなカテゴリは、局所的に小さいと呼ばれる。約束通り、このような微妙な点はほとんど無視するつもりだが、記録のために触れておこうと思った。カテゴリー理論における多くの興味深い現象は、hom-set の要素が、合成の規則に従ったモルヒズムとしても、セットの点としても見ることができることに根源がある。ここで、 におけるモルヒズムの合成は、集合 （ L1_1D45A, L1_1D45A）におけるモノイダル積に変換される。

3.6 Challenges

1. Generate a free category from:
   (a) A graph with one node and no edges
   (b) A graph with one node and one (directed) edge (hint: this edge can be composed with itself)
   (c) A graph with two nodes and a single arrow between them
   (d) A graph with a single node and 26 arrows marked with the letters of the alphabet: a, b, c ... z.
2. What kind of order is this?
   (a) A set of sets with the inclusion relation: A is included in B if every element of A is also an element of B.
   (b) C++ types with the following subtyping relation: `T1` is a subtype of `T2` if a pointer to `T1` can be passed to a function that expects a pointer to `T2` without triggering a compilation error.
3. Considering that `Bool` is a set of two values `True` and `False`, show that it forms two (set-theoretical) monoids with respect to, respectively, operator `&&` (AND) and `||` (OR).
4. Represent the `Bool` monoid with the AND operator as a category: List the morphisms and their rules of composition.
5. Represent addition modulo 3 as a monoid category.

3.6 Challenges

1. 自由なカテゴリを生成する。
   (a) 一つのノードと辺を持たないグラフ (b) 一つのノードと一つの（有向）辺を持つグラフ（ヒント：この辺はそれ自身と合成できる） (c) 二つのノードとそれらの間に一つの矢印を持つグラフ (d) 一つのノードとアルファベットのa、b、c ... zで示された26個の矢印を持つグラフ。
2. What kind of order is this?
   (a) 包含関係を持つ集合： ☒は、☒のすべての要素が☒の要素でもあるとき☒に包含される。(b) C++の型であり、以下の部分型付け関係を持つ。T1へのポインタが、T2へのポインタを期待する関数にコンパイルエラーを起こさずに 渡せる場合、T1はT2のサブタイプである。
3. BoolはTrueとFalseの2つの値の集合であり、演算子&&(AND)と ||(OR)に対してそれぞれ2つの（集合論的） モノイドを形成することを示しなさい。4. AND演算子を持つBoolモノイドをカテゴリーとして表現しなさい。モルヒズムとその構成則を列挙せよ。5. 加算モジュロ3をモノイドカテゴリとして表せ。