2

Types and Functions

The category of types and functions plays an important role in programming, so let's talk about what types are and why we need them.

2.1 Who Needs Types?

There seems to be some controversy about the advantages of static vs. dynamic and strong vs. weak typing. Let me illustrate these choices with a thought experiment. Imagine millions of monkeys at computer keyboards happily hitting random keys, producing programs, compiling, and running them.

2

Types and Functions

型と関数というカテゴリーは、プログラミングにおいて重要な役割を果たします。

2.1 Who Needs Types?

静的型付けと動的型付け、強い型付けと弱い型付けの利点について、いくつかの論争があるようです。これらの選択肢を思考実験で説明しましょう。コンピュータのキーボードに向かって、何百万匹もの猿が楽しげにランダムなキー を打ち、プログラムを作り、コンパイルし、実行している様子を想像してください。

With machine language, any combination of bytes produced by monkeys would be accepted and run. But with higher level languages, we do appreciate the fact that a compiler is able to detect lexical and grammatical errors. Lots of monkeys will go without bananas, but the remaining programs will have a better chance of being useful. Type checking provides yet another barrier against nonsensical programs. Moreover, whereas in a dynamically typed language, type mismatches would be discovered at runtime, in strongly typed statically checked languages type mismatches are discovered at compile time, eliminating lots of incorrect programs before they have a chance to run.

So the question is, do we want to make monkeys happy, or do we want to produce correct programs?

The usual goal in the typing monkeys thought experiment is the production of the complete works of Shakespeare. Having a spell checker and a grammar checker in the loop would drastically increase the odds. The analog of a type checker would go even further by making sure that, once Romeo is declared a human being, he doesn't sprout leaves or trap photons in his powerful gravitational field.

機械語では、猿が作ったどんなバイトの組み合わせも受け入れられ、実行されます。しかし、高級言語では、コンパイラが字句や文法の誤りを検出できることがありがたいのです。多くのサルがバナナを食べずに帰ることになるが、残ったプログラムが有用になる可能性は高くなる。型チェックは、無意味なプログラムに対するもう一つの障壁を提供する。さらに、動的型付け言語では、型の不一致は実行時に発見されるのに対し、強い型付け静的チェック言語では、型の不一致はコンパイル時に発見され、多くの間違ったプログラムを実行する前に排除することができます。そこで問題なのは、サルを幸せにしたいのか、それとも正しいプログラムを作りたいのか、ということだ。タイピングをする猿の思考実験における通常の目標は、シェークスピア全集の制作である。スペルチェッカーと文法チェッカーをループに入れることで、その確率は飛躍的に高まります。タイプチェッカーのアナログは、ロミオが人間であると宣言された後、彼が葉を生やしたり、強力な重力場で光子を閉じ込めたりしないことを確認することで、さらに前進します。

2.2 Types Are About Composability

Category theory is about composing arrows. But not any two arrows can be composed. The target object of one arrow must be the same as the source object of the next arrow. In programming we pass the results of one function to another. The program will not work if the target function is not able to correctly interpret the data produced by the source function. The two ends must fit for the composition to work. The stronger the type system of the language, the better this match can be described and mechanically verified.

The only serious argument I hear against strong static type checking is that it might eliminate some programs that are semantically correct. In practice, this happens extremely rarely and, in any case, every language provides some kind of a backdoor to bypass the type system when that's really necessary. Even Haskell has `unsafeCoerce`. But such devices should be used judiciously. Franz Kafka's character, Gregor Samsa, breaks the type system when he metamorphoses into a giant bug, and we all know how it ends.

Another argument I hear a lot is that dealing with types imposes too much burden on the programmer. I could sympathize with this sentiment after having to write a few declarations of iterators in C++ myself, except that there is a technology called *type inference* that lets the compiler deduce most of the types from the context in which they are used. In C++, you can now declare a variable `auto` and let the compiler figure out its type.

In Haskell, except on rare occasions, type annotations are purely optional. Programmers tend to use them anyway, because they can tell a lot about the semantics of code, and they make compilation errors easier to understand. It's a common practice in Haskell to start a project by

2.2 型はコンポーザビリティのためにある

カテゴリ理論とは矢を構成することである。しかし、どんな2本の矢でも合成できるわけではありません。ある矢印の対象オブジェクトは、次の矢印の元オブジェクトと同じでなけれ ばなりません。プログラミングでは、ある関数の結果を別の関数に渡します。ターゲットとなる関数が、ソースとなる関数が生成したデータを正しく解釈できなければ、プログラムはうまくいきません。合成がうまくいくためには、両端がフィットしていなければならない。言語の型システムが強力であればあるほど、この一致をうまく記述し、機械的に検証することができます。強力な静的型チェックに対する唯一の深刻な反論は、意味論的に正しいプログラムを排除してしまうかもしれないということです。しかし、実際にはこのようなことはほとんどなく、どの言語でも本当に必要なときには型システムを迂回するための裏技が用意されています。HaskellにさえもunsafeCoerceがある。しかし、このような装置は慎重に使用されるべきである。フランツ・カフカの登場人物であるグレゴール・ザムザは、巨大なバグに変身するときに型システムを破ってしまうが、その結末は皆知っている通りだ。もう一つよく耳にするのは、型を扱うとプログラマーに負担がかかりすぎるという意見だ。私自身、C++でイテレータの宣言を何度か書いた経験があるので、この意見には共感できるのだが、型推論という技術があり、コンパイラが使用される文脈からほとんどの型を推論してくれるようになっている。C++では、変数autoを宣言して、コンパイラにその型を割り出させることができるようになった。Haskellでは、ごく稀な場合を除いて、型注釈は純粋にオプションである。なぜなら、コードのセマンティクスについて多くのことを教えてくれるし、コンパイルエラーを理解しやすくしてくれるからだ。Haskellでは、プロジェクトを開始するときに

designing the types. Later, type annotations drive the implementation and become compiler-enforced comments.

Strong static typing is often used as an excuse for not testing the code. You may sometimes hear Haskell programmers saying, “If it compiles, it must be correct.” Of course, there is no guarantee that a type-correct program is correct in the sense of producing the right output. The result of this cavalier attitude is that in several studies Haskell didn’t come as strongly ahead of the pack in code quality as one would expect. It seems that, in the commercial setting, the pressure to fix bugs is applied only up to a certain quality level, which has everything to do with the economics of software development and the tolerance of the end user, and very little to do with the programming language or methodology. A better criterion would be to measure how many projects fall behind schedule or are delivered with drastically reduced functionality.

As for the argument that unit testing can replace strong typing, consider the common refactoring practice in strongly typed languages: changing the type of an argument of a particular function. In a strongly typed language, it’s enough to modify the declaration of that function and then fix all the build breaks. In a weakly typed language, the fact that a function now expects different data cannot be propagated to call sites. Unit testing may catch some of the mismatches, but testing is almost always a probabilistic rather than a deterministic process. Testing is a poor substitute for proof.

2.3 What Are Types?

The simplest intuition for types is that they are sets of values. The type `Bool` (remember, concrete types start with a capital letter in Haskell) is

Haskellでは、プロジェクトを型設計から始めるのが一般的です。その後、型アノテーションが実装を動かし、コンパイラが強制するコメントとなる。強力な静的型付けは、しばしばコードをテストしない言い訳として使われます。Haskellのプログラマが、”コンパイルできるなら、正しいに違いない ”と言っているのを聞くことがある。もちろん、型付けが正しいプログラムが、正しい出力を生成するという意味で正しいという保証はない。このような軽率な態度の結果、いくつかの調査では、Haskellはコード品質で期待されるほど強力に先行することはなかった。商業的な環境では、バグ修正の圧力はある一定の品質レベルまでしかかからないようです。この品質レベルは、ソフトウェア開発の経済性やエンドユーザーの許容範囲に関係し、プログラミング言語や方法論にはほとんど関係しません。より良い基準は、どれだけのプロジェクトがスケジュールから遅れたり、機能が大幅に削減された状態で納品されたかを測定することでしょう。ユニットテストが強い型付けを置き換えることができるという議論については、強い型付け言語における一般的なリファクタリングの実践を考えてみてください：特定の関数の引数の型を変更することです。強い型付け言語では、その関数の宣言を修正し、ビルドの中断をすべて修正すれば十分です。弱い型付け言語では、関数が異なるデータを期待するようになったという事実は、呼び出し元には伝わらない。ユニットテストはミスマッチの一部を検出するかもしれませんが、テストはほとんどの場合、決定論的というよりむしろ確率論的なプロセスです。テストは証明の代用にはならないのです。

2.3 What Are Types?

型に対する最も単純な直感は、型は値の集合であるということです。Bool 型（Haskell では具象型は大文字で始まることを覚えておいてください）は、次のようなものです。

a two-element set of `True` and `False`. Type `Char` is a set of all Unicode characters like `a` or `ą`.

Sets can be finite or infinite. The type of `String`, which is a synonym for a list of `Char`, is an example of an infinite set.

When we declare `x` to be an `Integer`:

```
x :: Integer
```

we are saying that it's an element of the set of integers. `Integer` in Haskell is an infinite set, and it can be used to do arbitrary precision arithmetic. There is also a finite-set `Int` that corresponds to machine type, just like the C++ `int`.

There are some subtleties that make this identification of types and sets tricky. There are problems with polymorphic functions that involve circular definitions, and with the fact that you can't have a set of all sets; but as I promised, I won't be a stickler for math. The great thing is that there is a category of sets, which is called **Set**, and we'll just work with it. In **Set**, objects are sets and morphisms (arrows) are functions.

Set is a very special category, because we can actually peek inside its objects and get a lot of intuitions from doing that. For instance, we know that an empty set has no elements. We know that there are special one-element sets. We know that functions map elements of one set to elements of another set. They can map two elements to one, but not one element to two. We know that an identity function maps each element of a set to itself, and so on. The plan is to gradually forget all this information and instead express all those notions in purely categorical terms, that is in terms of objects and arrows.

In the ideal world we would just say that Haskell types are sets and Haskell functions are mathematical functions between sets. There is just one little problem: A mathematical function does not execute any code

真と偽の2つの要素からなる集合です。Char 型は a や ą のようなすべての Unicode 文字の集合です。集合は有限でも無限でもかまいません。Charのリストの同義語であるStringの型は、無限集合の例です。xをIntegerと宣言した場合.

```
x :: Integer
```

と宣言するとき、それは整数の集合の要素であると言っているのです。HaskellのIntegerは無限集合であり、これを用いて任意の精度の演算を行うことができる。また，C++のintと同じように，機械型に対応する有限集合Intもあります．この型と集合の識別には、いくつかの微妙な問題がある。循環的な定義を含むポリモーフィック関数や、すべての集合の集合を持つことができないという問題がありますが、約束したように、私は数学に固執するつもりはありません。素晴らしいことに、□□と呼ばれる集合のカテゴリがあり、私たちはそれを使って仕事をするだけなのです。□では、対象は集合で、形態素（矢印）は関数です。□　は非常に特殊なカテゴリで、実際にそのオブジェクトの中を覗くことができ、それによって多くの直感を得ることができるからです。例えば、空集合には要素がないことが分かっている。特殊な1要素集合があることも知っている。関数は、ある集合の要素を別の集合の要素にマッピングすることも知っている。関数は2つの要素を1つに対応させることはできますが、1つの要素を2つに対応させることはできません。恒等式関数は集合の各要素をそれ自身に写すことも知っている．このような情報を徐々に忘れていき、代わりにこれらの概念を純粋にカテゴリー的な用語で、つまり、オブジェクトと矢印で表現することを計画している。理想的には、Haskellの型は集合であり、Haskellの関数は集合間の数学的関数であると言うだけである。数学的関数はどんなコードも実行しません。

— it just knows the answer. A Haskell function has to calculate the answer. It's not a problem if the answer can be obtained in a finite number of steps — however big that number might be. But there are some calculations that involve recursion, and those might never terminate. We can't just ban non-terminating functions from Haskell because distinguishing between terminating and non-terminating functions is undecidable — the famous halting problem. That's why computer scientists came up with a brilliant idea, or a major hack, depending on your point of view, to extend every type by one more special value called the *bottom* and denoted by `_|_`, or Unicode ⊥. This "value" corresponds to a non-terminating computation. So a function declared as:

```
f :: Bool -> Bool
```

may return `True`, `False`, or `_|_`; the latter meaning that it would never terminate.

Interestingly, once you accept the bottom as part of the type system, it is convenient to treat every runtime error as a bottom, and even allow functions to return the bottom explicitly. The latter is usually done using the expression `undefined`, as in:

```
f :: Bool -> Bool
f x = undefined
```

This definition type checks because `undefined` evaluates to bottom, which is a member of any type, including `Bool`. You can even write:

```
f :: Bool -> Bool
f = undefined
```

(without the `x`) because the bottom is also a member of the type `Bool -> Bool`.

– 答えを知っているだけです。Haskellの関数は答えを計算しなければならないのです。その答えが有限のステップ数で得られるのであれば、その数がどんなに大きくても問題ありません。しかし、計算の中には再帰を伴うものもあり、その場合は決して終了しないかもしれません。Haskellでは、終了する関数と終了しない関数を区別することが決定不可能であるため、終了しない関数を禁止することはできません（有名なハルティング問題です）。そこで、コンピュータ科学者たちは素晴らしいアイデアを思いついたのですが、見方によっては大ハックです。この「値」は非終端計算に対応する。だから、次のように宣言された関数は

```
f :: Bool -> Bool
```

と宣言された関数は、真、偽、または_|_を返すかもしれません；後者は、決して終了しないことを意味します。興味深いことに、ひとたび底を型システムの一部として受け入れると、すべての実行時エラーを底として扱うことができ、さらに関数が底を明示的に返すこともできるようになる。後者は通常、以下のようにundefinedという表現を使って行われる。

```
f :: Bool -> Bool
f x = undefined
```

この定義では，undefinedはbottomと評価され，Boolを含むあらゆる型のメンバであるため，型チェックが行われます．と書くこともできます。

```
f :: Bool -> Bool
f = undefined
```

と書くこともできます(x はなし)．なぜなら bottom も Bool -> Bool 型のメンバであるからです．

Functions that may return bottom are called partial, as opposed to total functions, which return valid results for every possible argument.

Because of the bottom, you'll see the category of Haskell types and functions referred to as **Hask** rather than **Set**. From the theoretical point of view, this is the source of never-ending complications, so at this point I will use my butcher's knife and terminate this line of reasoning. From the pragmatic point of view, it's okay to ignore non-terminating functions and bottoms, and treat **Hask** as bona fide **Set**.[1]

2.4 Why Do We Need a Mathematical Model?

As a programmer you are intimately familiar with the syntax and grammar of your programming language. These aspects of the language are usually described using formal notation at the very beginning of the language spec. But the meaning, or semantics, of the language is much harder to describe; it takes many more pages, is rarely formal enough, and almost never complete. Hence the never ending discussions among language lawyers, and a whole cottage industry of books dedicated to the exegesis of the finer points of language standards.

There are formal tools for describing the semantics of a language but, because of their complexity, they are mostly used with simplified academic languages, not real-life programming behemoths. One such tool called *operational semantics* describes the mechanics of program execution. It defines a formalized idealized interpreter. The semantics of industrial languages, such as C++, is usually described using informal operational reasoning, often in terms of an "abstract machine."

[1]Nils Anders Danielsson, John Hughes, Patrik Jansson, Jeremy Gibbons, Fast and Loose Reasoning is Morally Correct. This paper provides justification for ignoring bottoms in most contexts.

底を返す可能性のある関数は、可能なすべての引数に対して有効な結果を返す全関数とは対照的に、部分関数と呼ばれます。底があるため、Haskellの型と関数のカテゴリは□□ではなくHaskと呼ばれることが多いでしょう。理論的な観点からは、これは終わりのない混乱の元なので、この時点で肉切り包丁を使って、この推論を打ち切ろうと思います。実用的な観点からは、非終端関数やボトムアップを無視して、Haskを正真正銘のŭ□として扱っても構わないのです。1

2.4 なぜ数学的モデルが必要なのか?

プログラマーとして、あなたはプログラミング言語の構文と文法に親しんでいます。言語のこれらの側面は、通常、言語仕様の一番最初に形式的な記法を用いて記述され ています。しかし、言語の意味、つまりセマンティクスを記述するのはもっと難しく、もっと多くのページが必要で、十分に形式的であることは稀であり、ほとんど完全であることはありません。そのため、言語専門家の間では終わりのない議論が交わされ、言語標準の細かい点の解釈に特化した本が家内工業的に出版されているのです。言語の意味論を記述するための正式なツールもありますが、複雑なため、現実の巨大なプログラミング言語ではなく、簡略化された学術言語に使われることがほとんどです。オペレーショナル・セマンティクスと呼ばれるこのようなツールは、プログラムの実行の仕組みを記述するものである。これは、形式的に理想化されたインタプリタを定義するものである。C++のような産業用言語の意味論は、通常、非公式な操作推論を用いて記述され、しばしば ”抽象機械 ”という言葉で表現される。

Anders Danielsson, John Hughes, Patrik Jansson, Jeremy Gibbons, Fast and Loose Reasoning is Morally Correct（速く緩い推論は道徳的に正しい）。この論文は、ほとんどの文脈で底辺を無視することの正当性を示しています。

The problem is that it's very hard to prove things about programs using operational semantics. To show a property of a program you essentially have to "run it" through the idealized interpreter.

It doesn't matter that programmers never perform formal proofs of correctness. We always "think" that we write correct programs. Nobody sits at the keyboard saying, "Oh, I'll just throw a few lines of code and see what happens." We think that the code we write will perform certain actions that will produce desired results. We are usually quite surprised when it doesn't. That means we do reason about programs we write, and we usually do it by running an interpreter in our heads. It's just really hard to keep track of all the variables. Computers are good at running programs — humans are not! If we were, we wouldn't need computers.

But there is an alternative. It's called *denotational semantics* and it's based on math. In denotational semantics every programming construct is given its mathematical interpretation. With that, if you want to prove a property of a program, you just prove a mathematical theorem. You might think that theorem proving is hard, but the fact is that we humans have been building up mathematical methods for thousands of years, so there is a wealth of accumulated knowledge to tap into. Also, as compared to the kind of theorems that professional mathematicians prove, the problems that we encounter in programming are usually quite simple, if not trivial.

Consider the definition of a factorial function in Haskell, which is a language quite amenable to denotational semantics:

```
fact n = product [1..n]
```

The expression `[1..n]` is a list of integers from `1` to `n`. The function `product` multiplies all elements of a list. That's just like a definition of factorial taken from a math text. Compare this with C:

問題は、オペレーショナル・セマンティクスを使ってプログラムに関することを 証明するのが非常に難しいということである。プログラムの性質を示すには、本質的に、理想化されたインタプリタに よってプログラムを「実行」しなければならない。プログラマーが正しさの正式な証明を行わないことは問題ではない。私たちは常に正しいプログラムを書いていると「思って」いる。誰もキーボードの前に座って、"ああ、ちょっとコードを投げて何が起こるか見てみよう "なんてことはしない。私たちは、自分が書いたコードが特定の動作を行い、望ましい結果をもたらすと考えています。そうでないときは、たいていの場合、かなり驚きます。つまり、私たちは自分の書いたプログラムについて推論しているのですが、それはたいてい頭の中でインタプリタを走らせることで行っているのです。ただ、すべての変数を把握するのは本当に大変なんです。コンピュータはプログラムを実行するのが得意ですが、人間はそうではありません。もしそうなら、コンピュータは必要ないでしょう。しかし、別の方法があります。それは「含意論」と呼ばれるもので、数学に基づいている。デノテーショナル・セマンティクスでは、プログラミングの各構文を数学的に解釈する。これによって、プログラムの性質を証明したければ、数学の定理を証明すればよいことになる。定理を証明するのは難しいと思われるかもしれませんが、実は人間は何千年も前から数学的手法を積み重ねてきており、利用できる知識は豊富に蓄積されているのです。また、プロの数学者が証明するような定理に比べれば、プログラミングで遭遇する問題は、些細なものでないにしても、極めてシンプルであることがほとんどです。Haskellの階乗関数の定義を考えてみよう。Haskellは含意論的意味論に非常に適した言語である。

```
fact n = product [1..n]
```

1..n] という式は，1 から n までの整数のリストである．関数productはリストのすべての要素を乗算する．これは，数学の教科書に載っている階乗の定義のようなものです．これをC言語と比較してみましょう。

```
int fact(int n) {
    int i;
    int result = 1;
    for (i = 2; i <= n; ++i)
        result *= i;
    return result;
}
```

Need I say more?

Okay, I'll be the first to admit that this was a cheap shot! A factorial function has an obvious mathematical denotation. An astute reader might ask: What's the mathematical model for reading a character from the keyboard or sending a packet across the network? For the longest time that would have been an awkward question leading to a rather convoluted explanation. It seemed like denotational semantics wasn't the best fit for a considerable number of important tasks that were essential for writing useful programs, and which could be easily tackled by operational semantics. The breakthrough came from category theory. Eugenio Moggi discovered that computational effect can be mapped to monads. This turned out to be an important observation that not only gave denotational semantics a new lease on life and made pure functional programs more usable, but also shed new light on traditional programming. I'll talk about monads later, when we develop more categorical tools.

One of the important advantages of having a mathematical model for programming is that it's possible to perform formal proofs of correctness of software. This might not seem so important when you're writing consumer software, but there are areas of programming where the price of failure may be exorbitant, or where human life is at stake. But even when writing web applications for the health system, you may


```
int fact(int n) {
    int i;
    int result = 1;
    for (i = 2; i <= n; ++i)
        result *= i;
    return result;
}
```

これ以上言うことはないでしょう？さて、最初に断っておきますが、これは安直な表現です。階乗関数は明らかに数学的な意味を持っています。キーボードから文字を読んだり、ネットワークにパケットを送ったりするときの数学的モデルは何だろう？長い間、この質問は、かなり複雑な説明につながる厄介な質問だっただろう。デノテーショナル・セマンティクスは、有用なプログラムを書くために不可欠であり、オペレーショナル・セマンティクスで容易に取り組むことができる、かなりの数の重要なタスクに最適ではないように思えたのである。その突破口となったのは、カテゴリー理論である。エウジェニオ・モッジは、計算効果がモナドに写像できることを発見した。これは、意味論に新しい息吹を与え、純粋関数型プログラムをより使いやすくしただけでなく、従来のプログラミングに新しい光を当てる重要な観察であることが判明した。モナドについては、後ほど、よりカテゴリカルなツールを開発したときにお話しすることにします。プログラミングのための数学的モデルを持つことの重要な利点の一つは、ソフトウェアの正しさの形式的な証明を行うことができることです。消費者向けのソフトウェアを書いているときには、これはそれほど重要ではないように思えるかもしれませんが、プログラミングの分野では、失敗の代償が法外であったり、人命がかかっていたりすることがあるのです。しかし、医療システム用のWebアプリケーションを書くときでさえ、次のようなことがあります。

appreciate the thought that functions and algorithms from the Haskell standard library come with proofs of correctness.

2.5 Pure and Dirty Functions

The things we call functions in C++ or any other imperative language, are not the same things mathematicians call functions. A mathematical function is just a mapping of values to values.

We can implement a mathematical function in a programming language: Such a function, given an input value will calculate the output value. A function to produce a square of a number will probably multiply the input value by itself. It will do it every time it's called, and it's guaranteed to produce the same output every time it's called with the same input. The square of a number doesn't change with the phases of the Moon.

Also, calculating the square of a number should not have a side effect of dispensing a tasty treat for your dog. A "function" that does that cannot be easily modelled as a mathematical function.

In programming languages, functions that always produce the same result given the same input and have no side effects are called *pure functions*. In a pure functional language like Haskell all functions are pure. Because of that, it's easier to give these languages denotational semantics and model them using category theory. As for other languages, it's always possible to restrict yourself to a pure subset, or reason about side effects separately. Later we'll see how monads let us model all kinds of effects using only pure functions. So we really don't lose anything by restricting ourselves to mathematical functions.

Haskellの標準ライブラリの関数やアルゴリズムが正しいという証明付きで提供されていることに感謝する。

2.5 Pure and Dirty Functions

C++や他の命令型言語で関数と呼ばれているものは、数学者が関数と呼ぶものとは違います。数学的な関数とは、値から値への写像に過ぎません。数学的な関数は、プログラミング言語で実装することができます。このような関数は、入力値が与えられると、出力値を計算する。ある数値の2乗を計算する関数は、入力値を自分自身で掛け合わせるでしょう。呼び出されるたびにそれを行い、同じ入力で呼び出されるたびに同じ出力を生成することが保証されています。数の2乗は月の満ち欠けで変わることはない。また、数の2乗を計算すると、犬においしいおやつを配るという副作用があってはなりません。そんなことをする「関数」は、数学的な関数として簡単にモデル化することはできない。プログラミング言語では、同じ入力があれば常に同じ結果を出し、副作用がない関数を「純粋関数」と呼ぶ。Haskellのような純粋関数型言語では、すべての関数が純粋である。そのため、これらの言語には含意論的な意味づけを行い、カテゴリー理論を使ってモデル化することが容易である。他の言語では、純粋な部分集合に限定したり、副作用を別に推論したりすることが可能です。後で、モナドを使えば、純粋な関数だけを使って、あらゆる種類の効果をモデル化することができることを説明する。ですから、数学的な関数に限定しても、何も失うものはないのです。

2.6 Examples of Types

Once you realize that types are sets, you can think of some rather exotic types. For instance, what's the type corresponding to an empty set? No, it's not C++ `void`, although this type *is* called `Void` in Haskell. It's a type that's not inhabited by any values. You can define a function that takes `Void`, but you can never call it. To call it, you would have to provide a value of the type `Void`, and there just aren't any. As for what this function can return, there are no restrictions whatsoever. It can return any type (although it never will, because it can't be called). In other words it's a function that's polymorphic in the return type. Haskellers have a name for it:

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

(Remember, `a` is a type variable that can stand for any type.) The name is not coincidental. There is deeper interpretation of types and functions in terms of logic called the Curry-Howard isomorphism. The type `Void` represents falsity, and the type of the function `absurd` corresponds to the statement that from falsity follows anything, as in the Latin adage "ex falso sequitur quodlibet."

Next is the type that corresponds to a singleton set. It's a type that has only one possible value. This value just "is." You might not immediately recognize it as such, but that is the C++ `void`. Think of functions from and to this type. A function from `void` can always be called. If it's a pure function, it will always return the same result. Here's an example of such a function:

```
int f44() { return 44; }
```

You might think of this function as taking "nothing", but as we've just seen, a function that takes "nothing" can never be called because there is

2.6 Examples of Types

型が集合であることを理解すると、かなりエキゾチックな型を考えることができるようになります。例えば、空集合に対応する型は何でしょうか？HaskellではVoidと呼ばれる型ですが、これはC++のvoidではありません。HaskellではVoidと呼ばれるこの型は、値が存在しない型なのです。Voidを受け取る関数を定義することはできますが、それを呼び出すことはできません。呼び出すには、Void型の値を提供しなければならないのですが、それがないのです。この関数が何を返すかについては、何の制限もありません。どんな型でも返すことができる（ただし、呼び出すことはできないので、返すことはない）。言い換えれば、これは戻り値の型が多相性である関数なのだ。Haskellersはこの関数をこう呼んでいる。

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

（aは任意の型を表す型変数であることを忘れないでください）この名前は偶然ではありません。型と関数の論理的な解釈には、カレー・ハワード同型と呼ばれる深いものがある。Voidという型は虚偽を表し、absurdという関数の型は、ラテン語の格言 "ex falso sequitur quod libet" のように、虚偽から何でも続くという言葉に対応するものである。次に、シングルトン集合に対応する型である。これは、取りうる値が1つしかない型だ。この値は、ただ "ある "だけです。すぐにそうとはわからないかもしれませんが、これがC++のvoidです。この型からの関数と、この型への関数を考えてみてください。voidからの関数は、常に呼び出すことができます。純粋な関数であれば、常に同じ結果を返します。以下は、そのような関数の例です。

```
int f44() { return 44; }
```

この関数は「何も」受け取らないと思うかもしれませんが、今見たように、「何も」受け取らない関数は決して呼び出すことができません。

no value representing “nothing.” So what does this function take? Conceptually, it takes a dummy value of which there is only one instance ever, so we don’t have to mention it explicitly. In Haskell, however, there is a symbol for this value: an empty pair of parentheses, `()`. So, by a funny coincidence (or is it a coincidence?), the call to a function of void looks the same in C++ and in Haskell. Also, because of the Haskell’s love of terseness, the same symbol `()` is used for the type, the constructor, and the only value corresponding to a singleton set. So here’s this function in Haskell:

```
f44 :: () -> Integer
f44 () = 44
```

The first line declares that `f44` takes the type `()`, pronounced “unit,” into the type `Integer`. The second line defines `f44` by pattern matching the only constructor for unit, namely `()`, and producing the number 44. You call this function by providing the unit value `()`:

```
f44 ()
```

Notice that every function of unit is equivalent to picking a single element from the target type (here, picking the `Integer` 44). In fact you could think of `f44` as a different representation for the number 44. This is an example of how we can replace explicit mention of elements of a set by talking about functions (arrows) instead. Functions from unit to any type *A* are in one-to-one correspondence with the elements of that set *A*.

What about functions with the `void` return type, or, in Haskell, with the unit return type? In C++ such functions are used for side effects, but we know that these are not real functions in the mathematical sense of

というのは、「何もない」を表す値がないからです。では、この関数は何を受け取るのでしょうか。概念的には1つしか存在しないダミーの値を取るので、明示的に言及する必要はありません。しかし、Haskellでは、この値を表す記号として、空の括弧の組、()が存在します。つまり、不思議なことに（あるいは偶然なのか）、voidの関数の呼び出しはC++でもHaskellでも同じに見えるのです。また、Haskellは簡潔さを好むので、型、コンストラクタ、シングルトン集合に対応する唯一の値には同じ記号()が使われます。というわけで、この関数をHaskellで書くとこうなる。

```
f44 :: () -> Integer
f44 () = 44
```

最初の行は、f44が「ユニット」と発音される型()をInteger型に取り込むことを宣言しています。2行目は、unitの唯一のコンストラクタである()をパターンマッチして、数44を生成することで、f44を定義しています。この関数を呼び出すには、unitの値()を指定します。

```
f44 ()
```

unitのすべての関数は、対象となる型から1つの要素を選ぶことと同じであることに注意してください（ここでは、Integer 44を選んでいます）。実際、f44は数値44の別の表現と考えることができます。これは、集合の要素について明示的に言及する代わりに、関数（矢印）について語ることができる、という例です。単位から任意の型ŭへの関数は、集合ŭの要素と一対一で対応します。void 型の戻り値、Haskell では unit 型の戻り値の関数はどうでしょうか。C++では、このような関数は副作用のために使われますが、これらは数学的な意味での実関数でないことが分かっています。

the word. A pure function that returns unit does nothing: it discards its argument.

Mathematically, a function from a set A to a singleton set maps every element of A to the single element of that singleton set. For every A there is exactly one such function. Here's this function for `Integer`:

```
fInt :: Integer -> ()
fInt x = ()
```

You give it any integer, and it gives you back a unit. In the spirit of terseness, Haskell lets you use the wildcard pattern, the underscore, for an argument that is discarded. This way you don't have to invent a name for it. So the above can be rewritten as:

```
fInt :: Integer -> ()
fInt _ = ()
```

Notice that the implementation of this function not only doesn't depend on the value passed to it, but it doesn't even depend on the type of the argument.

Functions that can be implemented with the same formula for any type are called parametrically polymorphic. You can implement a whole family of such functions with one equation using a type parameter instead of a concrete type. What should we call a polymorphic function from any type to unit type? Of course we'll call it `unit`:

```
unit :: a -> ()
unit _ = ()
```

In C++ you would write this function as:

という意味での本当の関数ではないことがわかります。単位を返す純粋な関数は何もしません：その引数を破棄します。数学的には、集合 ☒ から単一集合への関数は、 ☒ のすべての要素を単一集合の単一の要素に写すものです。すべての ☒ に対して、そのような関数がひとつだけ存在する。以下は、この関数のIntegerに対するものである。

```
fInt :: Integer -> ()
fInt x = ()
```

この関数に任意の整数を与えると、単位が返されます。Haskellでは、簡潔さの精神から、破棄される引数にはアンダースコアというワイルドカードパターンを使うことができます。こうすれば、わざわざ名前を作らなくても済む。だから、上記は次のように書き換えることができる。

```
fInt :: Integer -> ()
fInt _ = ()
```

この関数の実装は、渡された値に依存しないだけでなく、引数の型にさえも依存しないことに注意してください。どんな型でも同じ式で実装できる関数をパラメトリックポリモーフィックと呼びます。このような関数のファミリ全体を，具象型の代わりに型パラメータを使って1つの式で実装することができます．任意の型から単位型へのポリモーフィックな関数を何と呼べばよいのでしょうか？もちろん、ユニットと呼ぶことにします．

```
unit :: a -> ()
unit _ = ()
```

C++では、この関数を次のように書きます。

```
template<class T>
void unit(T) {}
```

Next in the typology of types is a two-element set. In C++ it's called `bool` and in Haskell, predictably, `Bool`. The difference is that in C++ `bool` is a built-in type, whereas in Haskell it can be defined as follows:

```
data Bool = True | False
```

(The way to read this definition is that `Bool` is either `True` or `False`.) In principle, one should also be able to define a Boolean type in C++ as an enumeration:

```
enum bool {
    true,
    false
};
```

but C++ `enum` is secretly an integer. The C++11 "`enum class`" could have been used instead, but then you would have to qualify its values with the class name, as in `bool::true` and `bool::false`, not to mention having to include the appropriate header in every file that uses it.

Pure functions from `Bool` just pick two values from the target type, one corresponding to `True` and another to `False`.

Functions to `Bool` are called *predicates*. For instance, the Haskell library `Data.Char` is full of predicates like `isAlpha` or `isDigit`. In C++ there is a similar library that defines, among others, `isalpha` and `isdigit`, but these return an `int` rather than a Boolean. The actual predicates are defined in `std::ctype` and have the form `ctype::is(alpha, c)`, `ctype::is(digit, c)`, etc.


```
template<class T>
void unit(T) {}
```


型の類型の次は、2要素セットです。C++ではboolと呼ばれ、Haskellでは予想通りBoolです。C++ではboolは組み込み型ですが、Haskellでは次のように定義できる点が異なります。


```
data Bool = True | False
```


（この定義の読み方は、Bool は True か False のどちらかです。）原則的には、C++でも列挙型として Boolean 型を定義できるはずです。


```
enum bool {
    true,
    false
};
```


しかし、C++のenumは密かに整数である。しかし、C++11 の ”enum クラス ” を代わりに使うこともできますが、その場合は bool::true と bool::false のようにクラス名でその値を修飾しなければなりませんし、それを使うすべてのファイルに適切なヘッダーを含めなければならないのは言うまでもありません。Boolからの純粋な関数は、対象の型から2つの値、1つはTrueに、もう1つはFalseに対応するものを選ぶだけです。Boolに対する関数は述語と呼ばれます。例えば、HaskellのライブラリData.Charは、isAlphaやisDigitのような述語で一杯です。C++にも同様のライブラリがあり、isalphaやisdigitなどが定義されていますが、これらはブール値ではなく、int値を返します。実際の述語は std::ctype で定義され、 ctype::is(alpha, c) , ctype::is(digit, c) などの形式をとります。

2.7 Challenges

1. Define a higher-order function (or a function object) `memoize` in your favorite language. This function takes a pure function `f` as an argument and returns a function that behaves almost exactly like `f`, except that it only calls the original function once for every argument, stores the result internally, and subsequently returns this stored result every time it's called with the same argument. You can tell the memoized function from the original by watching its performance. For instance, try to memoize a function that takes a long time to evaluate. You'll have to wait for the result the first time you call it, but on subsequent calls, with the same argument, you should get the result immediately.
2. Try to memoize a function from your standard library that you normally use to produce random numbers. Does it work?
3. Most random number generators can be initialized with a seed. Implement a function that takes a seed, calls the random number generator with that seed, and returns the result. Memoize that function. Does it work?
4. Which of these C++ functions are pure? Try to memoize them and observe what happens when you call them multiple times: memoized and not.

   (a) The factorial function from the example in the text.

   (b) `std::getchar()`

   (c)
   ```
   bool f() {
       std::cout << "Hello!" << std::endl;
       return true;
   }
   ```

   (d)
   ```
   int f(int x) {
       static int y = 0;
   ```


2.7 Challenges

1. 高階の関数（または関数オブジェクト）memoize を好きな言語で定義する。この関数は純粋な関数 f を引数として取り、f とほとんど同じ振る舞いをする関数を返します。ただし、引数ごとに元の関数を一度だけ呼び、その結果を内部に保存し、その後同じ引数で呼び出されるたびにこの保存した結果を返します。メモ化された関数と元の関数を見分けるには、そのパフォーマンスを見ればよいでしょう。例えば、評価に時間がかかる関数をメモしてみましょう。最初に呼び出したときは結果を待つ必要がありますが、その後同じ引数で呼び出すと、すぐに結果が得られるはずです。2. 2. 標準ライブラリにある、乱数を生成するために通常使用する関数をメモ化してみてください。うまくいくか？3. 3.ほとんどの乱数生成器はシードで初期化することができます。種を受け取り、その種で乱数発生器を呼び出し、その結果を返す関数を実装してください。その関数をメモしてください。うまくいくか？4. これらの C++ 関数のうち、純粋なものはどれだろうか？それらをメモして、複数回呼び出したときに何が起こるかを観察してください： メモしたものとそうでないもの。

(a) テキストにある例の階乗関数。(b) std :: getchar() (c) bool f () {。

```
        std::cout << "Hello!" << std::endl;
        return true;
    }
```

(d)
```
int f(int x) {
    static int y = 0;
```

```
        y += x;
        return y;
    }
```

5. How many different functions are there from `Bool` to `Bool`? Can you implement them all?
6. Draw a picture of a category whose only objects are the types `Void`, `()` (unit), and `Bool`; with arrows corresponding to all possible functions between these types. Label the arrows with the names of the functions.


```
        y += x;
        return y;
    }
```

5. BoolからBoolへの関数は何種類ありますか？また、それらをすべて実装できますか？6. Void，()（単位），Bool の型と、これらの型の間で可能なすべての関数に対応する矢印を持つ、唯一の オブジェクトであるカテゴリの絵を描いてください。矢印には関数の名前を書きなさい。