Rigorous programming

```
rigour
/'rɪgə/
noun
    the quality of being extremely thorough and careful
```

See also *Intellectual rigour*.

The mathematician Terence Tao, in his blog post There's more to mathematics than rigour and proofs, describes the division of mathematical education into three stages:

1. The *pre-rigorous* stage.

   Mathematics is taught in an informal, intuitive manner, based on examples, fuzzy notions, and hand-waving.

2. The *rigorous* stage.

   One is taught that in order to do maths "properly", one needs to work and think in a much more precise and formal manner.

3. The *post-rigorous* stage.

   One has grown comfortable with all the rigorous foundations of one's chosen field, and is now ready to revisit and refine one's pre-rigorous intuition on the subject, but this time with the intuition solidly buttressed by rigorous theory.

   The point of rigour is not to destroy all intuition; instead, it should be used to destroy bad intuition while clarifying and elevating good intuition. It is only with a combination of both rigorous formalism and good intuition that one can tackle complex mathematical problems.

   The ideal state to reach is when every heuristic argument naturally suggests its rigorous counterpart, and vice versa. Then you will be able to tackle maths problems by using both halves of your brain at once – i.e., the same way you already tackle problems in "real life".

The very same thoughts apply with *mathematics* replaced by *computer science*, and, in particular, *computer programming*.

These lecture notes are about the *beginning* of a *rigorous* stage of *computer programming*, with the intention of leading to a post-rigorous stage in the above sense.

From pre-rigorous to rigorous programming

We write programs and we have an intuitive idea of what they are intended to do. We test them to make sure.

Example. Write a program to reverse a list. Sample solution:

```
rev :: [a] -> [a]
rev [] = []
rev (x:xs) = rev xs ++ [x]
```

We may test this and see that `rev [1,2,3]` evaluates to `[3,2,1]`, and take this as sufficient evidence of the correctness of our implementation of `rev`.

For this programming task, the pre-rigorous stage is perfectly fine and sufficient. It is easy to understand what is asked, and it is easy to see that the sample solution does do what is asked.

Moreover, for students without training in writing and reading rigorous proofs, such a proof will probably not be very enlightening. It will add little, if anything, to the understanding. It is only for more complex tasks that we need more rigour.

Program specifications

At the rigorous stage, we do proofs as suggested above, and we train students to read and write them. But before doing proofs, we need to understand rigorously *what* we are going to prove. This is the program *specification*. We want to give a convincing argument that the program satisfies its specification.

In the pre-rigorous stage, we still need to understand the *program specification*, which in the above example is the description of the task to be performed: to reverse a list.

If one doesn't know what "to reverse a list" means, an explanation is needed. One may give examples,

```
rev [] = []
rev [1,2,3] = [3,2,1]
```

and sometimes this is enough. If we want to be a little bit more precise, we may write the specification as

```
rev [a₀, a₁, ⋯, aₙ₋₂, aₙ₋₁] = [aₙ₋₁, aₙ₋₂, ⋯, a_1, a₀]
```

with the convention that when `n=0`, the expression `[a₀, a₁, ⋯, aₙ₋₂, aₙ₋₁]` stands for `[]`. (This is a common convention that is often not made explicit.)

If we want to be even more precise, we may say that what is required is that for all `i` in the range `[0 .. length n - 1]`, we have that

```
(rev xs) !! i = xs !! (length n - 1 - i)
```

where `(!! i)` picks the `i`th element of a list.

We may also give a *partial specification*, such as e.g.

```
rev (rev xs) = xs
```

which says that reversing a list twice gives the original list. This specification is partial (or *ambiguous*) because there are functions other than `rev` that satisfy this specification, like the identity function

```
id xs = xs
```

Another example of a partial specification is "find an element of a list satisfying a given property". The specification is partial because it doesn't say which element should be returned when there is more than one, or what should be done when there is none (should we use the `Maybe` type? should we give an error? or what?).

We can use types as parts of a specification, such as

```
find :: (a -> Bool) -> [a] -> a
find :: (a -> Bool) -> [a] -> Maybe a
```

Types by themselves, as in this example, also give partial specifications. More examples are

```
position  :: (a -> Bool) -> [a] -> Int
position  :: (a -> Bool) -> [a] -> Maybe Int
positions :: (a -> Bool) -> [a] -> [Int]
```

But the type alone is often not enough information of what is required, of course. We may add, e.g.

- Given a property of elements of a list, find a position where the element occurs in the list, giving an error if it doesn't occur.
- Given a property of elements of a list, find a position where the element occurs in the list, returning `-1` if it doesn't occur .

  This is probably a bad idea, but, good or bad, it is a specification.
- Given a property of elements of a list, find a position where the element occurs, using `Maybe` to account for the possibility that no element satisfying the property occurs in the list.
- Given a property of elements of a list, find the *first* position where the element occurs, using `Maybe` to account for the possibility that no element satisfying the property occurs in the list.
- Given a property of elements of a list, find the *last* position where the element occurs, using `Maybe` account for the possibility that no element satisfying the property occurs in the list.
- Given a property of elements of a list, find *all positions* where the element occurs, and collect them in a list (which will be empty if no element satisfying the property occurs in the given list).

Sometimes the name of a function may act as a partial specification, as in the above examples, but function names are not always wisely chosen and hence are not reliable in general. At best, function names act as reminders of what the functions specifications (to be found elsewhere) are, if the names are wisely chosen.

Example. The Hoogle specifications are usually stated in a *pre-rigorous* language. They are usually ambiguous, but often by accident rather than by explicit intent. And often they are not clear enough. The Hoogle specifications are still very useful. But often I find myself trying examples to attempt to clarify (seemingly undeliberate) ambiguities that matter for what I want to do.

One reason to give as precise specifications as possible is to avoid unwanted ambiguities. For example, the above two kinds of specifications `find` and `position` try to find something, but they are fundamentally different.

A third, also different, but related function is

```
occurs :: (a -> Bool) -> [a] -> Bool
```

with the specification that it tells whether or not an element satisfying a given property occurs in a given list

Formal specifications

All the above specifications are in the pre-rigorous stage, getting increasingly more rigorous. The ultimate stage of rigour is *formalism*, where we use a precisely defined language to write down computer program *specifications*, and also *proofs* that these specifications are satisfied.

Such formalisms were invented by mathematical logicians, and include first-order logic, set theory, Martin-Löf type theory, among others.

Computer scientists then implemented some such formal systems so that computers can check not only mathematics but also computer programs for correctness.

Formal specification and proof is an extreme form of rigour, where we not only give the specification in a formal language, but also prove that programs satisfy their intended specifications, with the proofs written also in this language, checked by the computer.

Learning objective of these lecture notes

Here our emphasis will be on *specifications* rather than *proofs*. That is, the question we address here is "how do we say rigorously, and even formally, what we intend our programs to do".

We emphasize that mathematics is usually done rigorously *but not formally* with computer checked proofs, although some people have formalized quite complicated mathematical proofs and checked them with computers.

Similarly, proofs of correctness of computer programs are usually done with paper and pencil (or even just in our heads) rather than formally on a computer, although again people have formally proved with computers the correctness of some tricky algorithms or of large pieces of software, including e.g. programming language compilers (whose correctness is definitely rather important).

Examples of rigorous computer-program specifications

We now discuss the problem of finding elements satisfying a given property in a list in order to illustrate the importance of clear specifications.

```
1. occurs    :: (a -> Bool) -> [a] -> Bool
2. find      :: (a -> Bool) -> [a] -> a
3. find      :: (a -> Bool) -> [a] -> Maybe a
4. position  :: (a -> Bool) -> [a] -> Int
5. position  :: (a -> Bool) -> [a] -> Maybe Int
6. positions :: (a -> Bool) -> [a] -> [Int]
```

As discussed above, the types already give an *indication* of what the functions are intended to do (that is, they give a *partial* or *ambiguous* specifications, in the sense discussed above).

1. Our intended specification for `occurs :: (a -> Bool) -> [a] -> Bool` is easy to state rigorously:

   Given a property `p :: a -> Bool` and a list `xs :: [a]`, we want that

   - `occurs p xs = True` if and only if there is `i :: Int` such that
     - `0 ≤ i < length xs` and
     - `p(xs !! i) = True`.

     This requirement says that there is a valid index `i` for the list `xs` such that the `i`th element of `xs` satisfies the property `p`. (It doesn't hurt repeating what the specification says in different words.)

   Even if this specification is precise, we may still give examples to aid understanding:

   - `occurs p [] = False` for any `p`.
   - `occurs even [1,3,5] = False`
   - `occurs even [1,2,3] = True`
   - `occurs odd [1,2,3] = True`
2. Our intended specification of `find :: (a -> Bool) -> [a] -> a` is almost as easy as that of `occurs`.

Given a property `p :: a -> Bool` and a list `xs :: [a]`, we want that

- *if* there is `i :: Int` such that `0 ≤ i < length xs` and `p (xs !! i) = True`
- *then* `find p xs` is some such element `xs !! i` of the list `xs`.

Notice that this is a conditional statement. If the condition is not satisfied, we are allowed to do anything we wish, such as giving an error, or returning any element of the list `xs` if the list is non-empty, etc.

Specifications are quite often conditional. In this case, the condition is often called the *pre-condition*. If the pre-condition is not satisfied, we are allowed to do anything we wish (compatible with the required type of the function, if a type is given as part of the specification).

An equivalent specification is the following:

Given a property `p :: a -> Bool` and a list `xs :: [a]`, we want that

- *if* there is `i :: Int` with `0 ≤ i < length xs` and `p (xs !! i) = True`,
- *then* `p (find p xs) = True` and also `find p xs` is an element of the list `xs`.

3. Our intended specification of `find :: (a -> Bool) -> [a] -> Maybe a` is similar.

   We want that `find p xs = Just x` implies that `x` occurs in the list `xs` and `p x = True`.

   We also want that `find p xs = Nothing` implies that there is no `x` that occurs in the list `xs` such that `p x = True`.

   Examples:

   - `find p [] = Nothing` for any `p`.
   - `find even [1,3,5] = Nothing`.
   - `find even [1,2,3] = Just 2`
   - `find odd [1,2,3] = Just 1`
   - `find odd [1,2,3] = Just 3`

   The reason the last two equations are possible (although not simultaneously, of course), is that our specification is ambiguous, or partial in the sense discussed above.

   When a specification is ambiguous, there may be different implementations satisfying the same specification. One implementation may satisfy `find odd [1,2,3] = Just 1` and another implementation may satisfy `find odd [1,2,3] = Just 3`. But, according to the specification we have given, even if it is ambiguous, any implementation must satisfy the first three equations.

   There is nothing intrinsically wrong with ambiguous specifications. Often specifications are deliberately ambiguous. (Like word auto-completion in a text editor. Which auto-completions should we suggest, when there is more than one? We may leave the decision open for the implementation to decide, without invalidating the specification.)

   But sometimes specifications are *accidentaly ambiguous*. Not because we wanted to be deliberately ambiguous, but because we were not precise enough in what we tried to say. Or maybe because we forgot to write down something we had in mind.

   Just as there may be incorrect programs (that is, programs that don't satisfy the intended specifications), there may be incorrect specifications (we had something in mind, but we didn't express it properly in words).

4. Our intended specification of `position :: (a -> Bool) -> [a] -> Int`.

   With the above background, we'll be brief this time.

   Given a property `p :: a -> Bool` and a list `xs :: [a]`, we want that

   - *if* there is `i :: Int` such that `0 ≤ i < length xs` and `p (xs !! i) = True`,
   - *then* `position p xs` is some such `i`.

   We are not saying what `position` should do if this condition is not fulfilled. Our specification is (deliberately or accidentally) ambiguous.

5. Our intended specification of `position :: (a -> Bool) -> [a] -> Maybe Int`.

   Given a property `p :: a -> Bool` and a list `xs :: [a]`, we want that

   - *if* there is `i :: Int` such that `0 ≤ i < length xs` and `p (xs !! i) = True`,
   - *then* `position p xs = Just i` for some such `i`.
   - *Otherwise*, we intend that `position p xs = Nothing`.

   We are not saying which such `i` should be returned in the case the pre-condition is fulfilled. Any such `i` is allowed (the first, the last, or an

arbitrary one).

6. Our intended specification of `positions :: (a -> Bool) -> [a] -> [Int]`.

   Given a property `p :: a -> Bool` and a list `xs :: [a]`, we want that

   - If some given `i :: Int` is in the list `positions p xs`, then `0 ≤ i < length xs` and `p (xs !! i) = True`.
   - If some given `i :: Int` with `0 ≤ i < length xs` satisfies the condition `p (xs !! i) = True`, then `i` is in the list `positions p xs`.

   Remark. The list `positions xs p` will be empty precisely when for all `x` in the list `xs` we have `p x = False`.

   I would say that this specification is pre-rigorous or post-rigorous, in the sense that when we read it we understand precisely what "is in the list" means.

   If we want to be fully rigorous, we can say:

   - For any `j :: Int` with `0 ≤ j < length (positions p xs)` we have that `p (xs !! (positions p xs !! j)) = True`.
   - If some given `i :: Int` with `0 ≤ i < length xs` satisfies the condition `p (xs !! i) = True`, then there is `j` with `0 ≤ j < length (positions p xs)` such that `positions p xs !! j = i`.

The language for rigorous specifications

It turns out that a limited logical vocabulary suffices, in practice, to write specifications.

- `for all`, `for some`, `and`, `or`, `not`, `implies`, `equals`.

A computer language for formal specifications and proofs

We will use a notation close to the Agda language for writing such formal specifications.

Formal versions of the above specifications

1. Specification of `occurs :: (a -> Bool) -> [a] -> Bool`:

```
∀ (a : Type) (p : a -> Bool) (xs : List a)

 → occurs p xs ≡ True ⇔ Σ i : Int , (0 ≤ i)
                                  × (i < length xs)
                                  × (p(xs !! i) ≡ True)
```

- Here `∀` means `for all` or `for any given` (universal quantification).
- The symbol `≡` stands for mathematical equality.
- The symbol `⇔` stands for logical equivalence `if and only if`.
- `Σ` (sum) plays the role of `there exists` or `for some` (existential quantification).
- `×` (cartesian product) plays the role of `and` (conjunction).

1. Specification of `find :: (a -> Bool) -> [a] -> a`

```
∀ (a : Type) (p : a -> Bool) (xs : List a)

 →  (Σ i : Int , (0 ≤ i)
               × (i < length xs)
               × (p (xs !! i) ≡ True))

       → (Σ i : Int , (0 ≤ i)
                    × (i < length xs)
                    × (p (xs !!  i) ≡ True)
                    × (find p xs ≡ xs !! i))
```

An equivalent specification is the following:

```
∀ (a : Type) (p : a -> Bool) (xs : List a)

 → (Σ i : Int , (0 ≤ i)
              × (i < length xs)
              × (p (xs !! i) ≡ True))
```

```
    → p (find p xs) ≡ True
    × (Σ i : Int , (0 ≤ i)
                 × (i < length xs)
                 × find p xs ≡ xs !! i)
```

2. Specification of `find :: (a -> Bool) -> [a] -> Maybe a`.

```
∀ (a : Type) (p : a -> Bool) (xs : List a)

 → (∀ (x : a) → find p xs ≡ Just x → (p x ≡ True)
                                   × (Σ i : Int , (0 ≤ i)
                                                × (i < length xs)
                                                × (xs !! i ≡ x)))

× (find p xs ≡ Nothing → ¬(Σ i : Int ,  (0 ≤ i)
                                    × (i < length xs)
                                    × p(xs !! i) ≡ True))
```

   - Here `¬` stands for `not` (negation).

3. Specification of `position :: (a -> Bool) -> [a] -> Int`.

```
∀ (a : Type) (p : a -> Bool) (xs : List a)

 →  (Σ i : Int , (0 ≤ i)
               × (i < length xs)
               × (p (xs !! i) ≡ True))

       → (Σ i : Int ,  (0 ≤ i)
                    × (i < length xs)
                    × (p (xs !! i) ≡ True)
                    × (position p xs ≡ i))
```

4. Specification of `position :: (a -> Bool) -> [a] -> Maybe Int`.

   Have a go, based on the above.

5. Our intended specification of `positions :: (a -> Bool) -> [a] -> [Int]`.

   Have a go, based on the above.

The Game of Life

What do "periodic" and "eventually periodic" mean.

Trees

Specifications and correctness proofs of some of the programs we've seen in lectures.

The monad laws

Are here.

Further reading

- Fast and Loose Reasoning is Morally Correct

I am not sure I subscribe to what the title claims, but at least this paper identifies some things we have (deliberately) overlooked in our discussion, which you should be aware of.

The issue is that we haven't discussed non-terminating programs, and this does affect how specifications should be written.