Models of Computation
Revision Notes

James Brown

April 26, 2017

Contents

1 Introduction

These are notes I have written in preparation for the upcoming 2017 Models of Computation exam. This year the module was run by Paul Levy (P.B.Levy@cs.bham.ac.uk). This module is about problems and *computers*. We ask ourselves:

- What problems can be solved on a computer?
- What problems can be solved on a computer with finitely many states?
- What problems can be solved on a computer with only finitely many states, but also a stack of unlimited size?
- What problems can be solved on a computer with only finitely many states, but also a tape of unlimited size that it can read and write to?
- What problems can be solved *fast* on a computer?
- What does "fast" mean anyway?
- What does *computer* mean anyway?

2 Language Membership Problems and Regular Expressions

2.1 Language Membership Problems

Suppose we have a set of characters Σ, which we will call the *alphabet*. A *word* is a finite sequence of characters, and we write Σ^* for the set of all words. We can *concatenate* words. A *language* is a set of words and a subset of Σ^*. Given a word, we want to know is it in the language or not? If we take an example alphabet a, b, c, here are some languages:

- All words which contain exactly 3 b's
- All words whose length is prime
- All words that have more b's than a's
- The words abc, bac and cb
- No words at all
- The empty word

These examples are largely pretty useless, but this problem does have real world applications such as

- Java has rules about what you can call a variable. Is the word read by the compiler a valid variable name?
- A user makes an account and enters a password, is it valid?
- A student has submitted code for an assignment, is it correct?
- Will this code crash when it's run?

In each one of these examples, we are provided with a word and we want to know whether it is an acceptable word. We want to make a computer tell us the answer.

2.2 Regex

Regular Expressions are a useful notation for describing languages. We write Empty for the language consisting of no words (the empty set). We write a for the language consisting of just the single-character a, and ϵ for the language consisting of just the empty word. If we have a language ***L*** and ***L'***, we write ***LL'*** for the set of words that are a concatenation of a word in ***L*** and a word in ***L'***. We can also write ***L*—*L'*** for the set of words that are in ***L*** or ***L'*** - the union of the two languages. We write ***L**** for the set of words that are a concatenation of some number of words in ***L*** (some number may be 0). Just like arithmetic, regular expressions have precedence rules. $*$ has the highest precedence, then juxtaposition and then $|$.

Regular expressions in theoretical computer science mean an expression built from the above operations and nothing more. Regular expressions are used much more widely in programming with far more operations, but they cannot be used in the module. For example, + is a very common operation but not available to us.

3 Finite State Automata

3.1 Removing ϵ moves

Let $(Q, \delta, \text{Acc}, q_0)$ be a NDFA with ϵ-transitions. There is a slow a-transition from state q to state r when starting from state q we can follow some number of ϵ-transistions, then an a-transition and end in state r. A state q is slowly accepting when starting from a state q we can follow some number of ϵ-transitions and reach and accepting state.

To create a NDFA without ϵ-transitions, we take the same states and the same initial state and make all slow transitions into normal transitions, and all slowly accepting states become accepting.

3.2 Determinisation

Given a NDFA (Q, δ, Acc, q_0) we can form a DFA. The states in the DFA will be subsets of Q. From a state $U \subseteq Q$, when we input a, we take the set of states that we can reach by an a-transition from a state in U. Our accepting states are states which contain a state which is contained in Acc. The initial state stays the same as before.

3.3 Coverting a regular expression to a deterministic finite state automaton

Simply create a NDFA from the regular expression, remove ϵ-moves and then determinise the automaton.

3.4 Converting a finite state automaton to a regular expression

4 Regular Languages

Theorem 1 (Kleene's Theorem) *A language is regular if and only if it is accepted by a finite state automaton.*

4.1 Operations on regular languages

We use Kleene's theorem to prove that some languages are regular. Firstly, the complement of a regular language L is also regular. To do this, we take the DFA $(Q, \delta, \text{Acc}, q_0)$ which accepts L. We then take the automaton $(Q, \delta, Q\ \text{Acc}, q_0)$. This automation recognises the complement of L, so the complement of L must also be a regular language.

Secondly, the intersection of regular languages L and L' is regular, as it is the complement of the union of the complements of L and L'.

4.2 Proving a language is not regular

Theorem 2 (Myhill-Nerode Theorem) *A language L is not regular if and only if it has an infinite distinctive sequence.*

Let w and w' be words, and L a language. An L-distinguishing suffix for w and w' is a word x such that either $wx \in L$ and $w'x \notin L$, or $wx \notin L$ and $w'x \in L$. For example, take the words `Wed` and `Tues`. An English-distinguishing suffix for these two words is `day` because `Wedday` is not and English word but `Tuesday` is.

Suppose we have a DFA which recognises the language L. If words w and w' have an L-distinguishing suffix, then w and w' must take us to different states from the inital state. Now suppose we have four words, w_0, w_1, w_2, w_3 that are **distinctive** (meaning they all have distinguishing suffixes in relation to each other. The DFA to represent this language must have at least four states. Now suppose there is an infinite sequence of words $w_0, w_1...$ that is **distinctive**. There cannot be a DFA to represent this language, so by Kleene's theorem the language is not regular.

Example: Take the language $\{a^n b^n \mid n \in \mathbb{N}\}$. If we take all a^n for each $n \in \mathbb{N}$, then for $m < n$, b^m is a distinguishing suffix for a^m and a^n. This is a distinctive infinite sequence, and the language is not regular.

4.3 Context Free Languages

5 Bisimulation and Minimisation

5.1 Bisimilarity

Two states are **bisimilar** if and only if they accept the same language. Suppose (Q, δ, Acc) and (Q', δ', Acc') are unpointed automata for the same alphabet L. A state $q_0 \in Q$ is bisimilar to a state $q_0' \in Q'$ if and only if they accept the same language. In order to prove this, let $\mathcal{R}$ be a bisimulation. We will prove that for any word w, if $q\mathcal{R}q'$ then q accepts w if and only if q' accepts w. This is by induction on w:

For the base case $w = \epsilon$:

$$\begin{aligned} & q \text{ accepts } \epsilon \\ \leftrightarrow\ & q \text{ is accepting} \\ \leftrightarrow\ & q' \text{ is accepting} \\ \leftrightarrow\ & q' \text{ accepts } \epsilon \end{aligned}$$

For the inductive step where w has a head (first character) a and a tail t:

$$\begin{aligned} & q \text{ accepts } a :: t \\ \leftrightarrow\ & \delta(q, a) \text{ accepts } t \\ \leftrightarrow\ & \delta'(q', a) \text{ accepts } t && \text{(by the inductive hypothesis)} \\ \leftrightarrow\ & q' \text{ accepts } a :: t \end{aligned}$$

The easiest way to construct a check for bisimilarity is to build a bisimulation (essentially a tree). From your pair of states, follow all outputs to new pairs of states. If any of the new pairs disagree on acceptance (one accepts and the other doesn't), then the original pair is not bisimilar as they both accept different languages. We stop building the tree down a branch if we see a pair of states that we have visited before, and conclude that the pair of states is bisimilar if we can't make any more moves that we haven't already seen before.

5.2 Minimisation

In order to minimise an automaton, there are two steps:

1. Throw away unreachable states
2. Identify bisimilar states and merge the ones that are bisimilar

The found minimal automaton will be unique (up to isomorphism).

6 The Halting Problem

In Computer Science there are many problems that we encounter. These all have an input (some string, a number, three strings etc.) and an output. Often this output is yes or no, but it may be a number or string or some other value. If a problem returns a yes or no result, then it is a decision problem. We have to draw a distinction between problems and instances. 'Is the word *abbab* accepted by the regexp $ab*(ab*|b)?$' is an instance of a more general problem 'Is a word w accepted by the regexp $ab*(ab*|b)?$'. Even more generally we could describe this problem as 'Given a word w and a regexp E, is w accepted by E?'.

Typically we want to consider problems which have a countably infinite set of acceptable inputs and acceptable outputs. Refer to Mathematical Techniques for Computer Science to see how we can ensure something is countably infinite. A problem can also be called a function, and we can also describe it as a subset of the set of inputs.

- Problem: given a positive integer, is it prime?
- The function that maps a positive integer to 'true' if it's prime and 'false' otherwise
- The subset of the set of positive integers consisting of all prime integers

If the set of inputs is the set of words over some alphabet, then the subset will be a language.

Key concepts:

- A decision problem (a problem which returns a yes or no answer) is decidable if it can be solved on a computer.
- A function is computable if it can be computed on a computer.

- A subset of the set of inputs (a language) is decidable if the corresponding decision problem can be solved on a computer.

We still haven't come up with a definition for *computer* yet! We get around this for now by saying a problem is 'Java-decidable' or 'Java-computable' for example, meaning that we can accomplish the task with a Java program.

6.1 Reducing a problem to another problem to solve the Halting problem

Suppose we have two problems - P and Q. If we can show how to solve Q using a black box which solves P then we can say that we have reduced the problem Q to the problem P. If we can reduce Q to P then if P is decidable, Q is decidable and if Q is undecidable, P is undecidable.

Given a nullary Java method such as the one below, can we tell if it will terminate or not? We keep things simple by assuming integers are unbounded, no exceptions are thrown and any such method when called either terminates or hangs forever).

```
void f (){
    ...
}
```

This is an example of the **Halting Problem**. Turing proved that the halting problem was undecidable. To prove this, we assume that it is decidable.

1. Consider the **unary halting problem**. Given a unary Java method

   ```
   void f (String x) {
       ...
   }
   ```

   and a string y, does `f` terminate when called with y? We reduce the unary halting problem to the nullary one. Given the unary method `f` and a string x, obtain a nullary method `g` by taking the code of `f` and replacing `x` with y - that is substitute in the argument values. Then, `g` terminates when called if `t` terminates when called with argument y. Since we assume the nullary halting problem is solvable, the unary one is too. This gives us a program

   ```
   boolean haltcheck (String somemethod, String y)
   ```

   where `somemethod` is the body of a unary method. When we apply with M and y this method returns true when M applied to y terminates, otherwise it returns false.

2. We build on this method further

   ```
   void hangcheck (String somemethod, String y) {
      if haltcheck(somemethod, y) {
         while true {}
      } else {
         return;
      }
   }
   ```

This method when applied to M and y, hangs if M applied to y terminates, otherwise it returns.

3. We build on this even further with a new program

```
void doublehang (String y) {
    haltcheck(y, y)
}
```

This method when applied to y (the body of the unary method), will hang if y applied to y terminates, otherwise it returns.

4. Finally, let z be the body of `doublehang`. We see that `doublehang`, when applied to z, terminates if and only if it hangs. This is a contradiction, so there cannot be any program which solves the halting problem.

7 Properties of Code

7.1 Rice's Theorem

Rice's theorem states that every non-trivial semantic property of code is undecidable. We define a trivial property is one which either holds for everything or holds for nothing.

To prove Rice's theorem, we let α be a non-trivial semantic property of code and let `alwayshang` be a piece of code that always hangs regardless of argument. We say that `alwayshang` doesn't satisfy α, and since α is non-trivial we can say that there is another piece of code C that does. For any nullary program P, we form $F(P)$ by inserting P at the start of the code C.

- If P terminates, then $F(P)$ has the same semantics as C. Because α is semantic and C satisfies α, $F(P)$ does too.
- If P hangs, then $F(P)$ has the same semantics as `alwayshang`. Since α is semantic and `alwayshang` doesn't satisfy α, $F(P)$ also doesn't.

In short, P will terminate if and only if $F(P)$ satisfies α. This means that if we can test for α, we can also test whether P terminates by testing if $F(P)$ satisfies α. We've already proven that we can't prove P terminates, so we must not be able to test for α - meaning α is undecidable.

8 Turing Machines

In 1936, Alan Turing invented the **turing machine**. Turing machines are very simple computers, and have a very limited set of instructions that they can execute. They have finitely many states, but also have an infinite tape which they can compute upon. Because of this, a Turing machine is an idealized computer and can never exist in the physical world. Turing machines are very convenient for analysing the running times of algorithms. Firstly, the time taken for an algorithm scales straightforwardly as the size of the input increases. They typically tend to treat inputs of size ten or one million in the same way. Secondly, every step of computation is made explicit in a very conservative way.

Turing machines have finitely many **states** and an infinite tape, with a **head** which sits over one space of the tape. They also have a finite tape alphabet T, and each space

on the tape contains a letter which is contained in T. T also contains the blank character, $\textvisiblespace \in T$, which is particularly important. All but finitely many spaces on the tape are blank - it never happens that infinitely many spaces contain actual, non-blank, characters. The two properties of a Turing machine - unlimited time and unlimited space - are essential.

Let T, the **tape alphabet** be a finite set of characters. Let R be a finite set of **return values**. This is commonly just a singleton set, which would be analagous to `void` in Java, and simply tells us whether the prorgam returns (halts, terminates) or runs forever. It may also be a larger set, like a two-element set analagous to `boolean` for example. A Turing machine over T and R consists for the following data:

- A finite set Q of states
- An initial state $q_0 \in Q$
- A transition function δ from Q to the following set of behaviours:
  - $(T \to Q)$ (read from current position)
  - $(T \times Q)$ (write to current position)
  - Q (move left)
  - Q (move right)
  - Q (do nothing)
  - R (stop)

9 Church's Thesis

Theorem 3 *What ever can be done by any computational device can be done by a Turing machine*

Church's thesis cannot be proven - it's entirely possible for someone to invent a new way of computation which extends the capabilities of programs as we know them. On the other hand, nobody has been able to come up with an alternative to Turing machines as of yet and the more time that passes the more plausible Church's thesis becomes.

10 Complexity and P

Until now we have been concerned with figuring out what a computer can and can't do. According to Church's thesis, this distinction of what we can and can't do does not depend at all on the architecture of the machine or the programming language - it's a fundamental divide. We have also seen that there are problems of practical intrest which exceed the capabilities of computers. These problems we can only solve approximately or partially. This section is concerned with classifying problems according to the computational effort they need for their solution. There are many different ways to accomplish this. In the simplest case we will look at decision problems which require a yes/no answer.

In computer science we use the term **complexity** differently than we do in English where it normally means 'multi-faceted' or 'confusing'. In the study of programs we use the word strictly to refer to the amount of resources needed during computation - primarily **time** and **space**. In select other cases we may be interested in other resources such as number of file accesses. We also must make the distinction between the complexity of an

algorithm (which we have studied before and refers to the runtime behaviour of a program) and the complexity of a problem which refers to an inherent difficulty in solving a problem.

We study the runtime of Turing machines rather than any real computer. This is for a number of reasons. Firstly, the time and Turing machine takes is unambiguously characterised by the number of transitions it makes allowing us to not have to bring in actual time measurements. Secondly, the space usage of a Turing machine can be defined unambiguously as well. We take this to be the number of cells it ever visits during computation. As tape cells have finite capacity only, this measure is fair and realistic. Thirdly, Turing machines operate uniformly on input of arbitrary size. A Turing machine will behave the same regardless of if a string is two characters long or two billion characters long which is not true of real hardware. Turing machines can scale up in size, real computer programs typically don't.

Once we have designed a Turing machine to solve a particular problem, how do we find the time complexity of the program? It is not sufficient to test the program on a few strings and just tabulate the outcomes. We would really like to give the number of steps executed by the machine for all possible inputs it can take - this is the function from the input domain (for example Σ^*, strings over Σ) to the set of natural numbers. Such a formula is probably very difficult to find for even the most simple Turing machines. It is also not clear whether we can read off a lot from the formula should we find it.

If we abstract away from the actual input to the length of the input, the situation becomes much more manageable both in terms of finding the formula and in terms of interpreting the given formula. Mathematically, we will get a function from the natural numbers (the input length) to the natural numbers (number of steps taken). It's important to be precise about how the abstraction is achieved. We may associate the input length n with n^2 if the Turing machine takes at most n^2 steps. This is very different to interpretting n^2 as the number of steps on average. We can distinguish three cases when analysing the runtime complexity of an algorithm:

- **Best-case runtime** or **Minimum time**: The time needed when the algorithm is applied to the 'easiest' possible input. An example is applying a sorting algorithm to the already sorted list.
- **Worst-case runetime** or **Maximum time**: The time needed when the algorithm is applied to the 'worst' possible input. An example is applying a sorting algorithm to a list sorted in reverse order.
- **Average-case runtime** or **Average time**: The average time needed (where the average is taken over all possible inputs).

We known how to determine the time-complexity of algorithms from first-year. We now take one further abstraction and ask whether problems themselves have an inherent complexity? In general the answer to this question is sometimes 'yes', sometimes 'yes but currently unknown' and sometimes 'no'. We are trying to make a statement about all algorithms which would solve the problem by assigning a complexity to a problem. This idea can fail in practice in a number of ways.

1. We can have a situation where we suspect that the problem is solvable but nobody has been able to give an algorithm for it yet.
2. We may have one or a number of algorithms but we are unable to determine their complexity functions. This was the case for the Simplex Method.

3. We may have a few algorithms and we may know their complexity but we do not know whether there exist better algorithms or not.

4. We may have the situation that there are probably better and better algorithms for a given problem, we can then never determine the largest lower bound of the complexity.

Cases 1 and 2 are rare but have some spectacular examples. Case 4 seems to only occur in problems that have been designed specifically for that purpose. Case 3 is the most common situation. The most common example of case 3 is that we know we can solve the Travelling Salesman problem in exponential time but we don't know if there exists a more efficient algorithm. In general, for every specific algorithm, whose runtime behaviour is known, gives us an *upper bound* for the complexity of a problem (like saying I know I can do this in f(n) many steps, but maybe we can do better). A *lower bound* cannot be established by programming at all, instead you need a clever way of analysing the problem which shows that every algorithm must take atleast so many steps. For many problems, we don't have lower bounds yet and they are very challeging to come across.

10.1 The complexity class $\mathcal{P}$

For the problem of copying a string on a one-tape Turing machine, we can show that every implementation will have a time complexity which is at least quadratic. However, when we change to a two tape scenario we can achieve a much faster implementation which runs in linear time. Because of this, should we religiously stick to one tape machines, or should we be more liberal in our choice of machine? What this example shows us is that we need to say which machine model we are referring to - by convention this is the multitape Turing machine.

Are there complexity classes which are independent of the machine model? Absolutely. Problems which have a polynomial complexity on Turing machines (problems whose complexity class is $O(n^k)$ have polynomial complexity on almost all machine models. A machine model is 'reasonable' only if it defines the same class of polynomial problems as Turing machines. The class of polynomial problems with polynomial implementations is denoted by $\mathcal{P}$ and it is customary to call the problems in $\mathcal{P}$ **feasible** and those outside **infeasible** or **intractable**.

11 NP

We have introduced the distinction between problems which have a solution which runs in polynomial time and problems for which a solution does not exist. The former problems are **feasible** and the latter are **intractable**. We use $\mathcal{P}$ to notate all problems which are feasible.

The distinction between $\mathcal{P}$ and not-$\mathcal{P}$ is not easy to establish in practice.

There is a whole range of very common problems which are not sure if they are feasible or not. These are called **$\mathcal{NP}$-complete problems**. In a nutshell, for $\mathcal{NP}$-complete problems we do not know whether the problems are really hard, but we have very good grounds to believe it.

The easiest way to explain the class $\mathcal{NP}$ is via the distinction between solving a problem and checking a solution. From experience, we know that checking a solution is much easier than solving a problem. $\mathcal{NP}$ consists of all the problems for which there exists a polynomial solution checking algorithm. For example, it is very easy to check that a

number is divided by another, no matter how large the numbers are. On the other hand, finding a factor of a given number - even if you know if the number is prime or not - is very hard to do. RSA encryption is built upon this fact, and solving the problem would have profound effects.

The formal definition of what it means to check a solution is a little involved. This is because each instance may require additional data to do the checking efficiently, having the answer alone may not be enough. We will look at decision problems where the answer is either 'yes' or 'no'. Such an answer does not make it any easier to check whether it is correct or not. We need something additional, which we call a **proof** or a **witness**. In the division example this would be the factors of the number (which gives us more information that just that the number is composite). The checker can then verify the answer by multiplying the factors together. We formalise this idea of a witness:

Definition 1 *A subset L of Σ^* belongs to $\mathcal{NP}$ if there exists a determinisitc Turing machine M which takes pairs (s, w) of strings as input and replies 'yes' or 'no' in polynomial time. The replies are to be interpretted as saying 'yes, w proves that s belongs to L' or 'no, this w is not a proof for the membership of s in L. Furthermore, it must be true that for every $s \in L$ there exists at least one witness $w \in \Sigma^*$ such that M will reply 'yes' when presented with (s, w) and for every $s \in L$ no witness exists for which M replies 'yes'.*

11.1 Nondeterministic Turing Machines

We can define the class $\mathcal{NP}$ is a completely different way using **nondeterministic Turing machines**. NTMs are defined just like determinisitic Turing machines expect we use a transition *relation* rather than a transition *function*. In essence, we get a transition table where every entry consists of finitely many choices for how the machine may proceed and the machine is allowed to choose freely at every stage one of the transistions from the available collection.

We will concentrate our efforts on decision problems once again. When we design an NTM for a decision problem it makes sense to allow for a third possible outcome: 'undecided' or 'can't make a judgement either way'. The machine is **correct** if it never replies 'yes' when it should have said 'no' and if it never replies 'no' when it should have said 'yes'. In all cases it is allowed to respond with 'undecided'. An NTM is said to **solve** a decision problem if it is correct and for every positive instance there is at least one computation path along which the machine will reply 'yes'.

We can use an NTM to construct a deterministic machine which truly decides the decision problem (although it takes a long time to do so). The determinisitic machine systematically explores all execution paths the nondeterministic machine is capable of. As the NTM is bound to stop after a fixed amount of time, there is only finitely many paths to consider. If one branch ends with 'yes' then the determinisitic machine answers 'yes' as well. If all branches end in 'undecided' or one branch answers 'no', the deterministic machine answers 'no'.

The worst case complexity of a NTM is defined almost as in the determinisitic case: we take the maximum run time (or memory usage) on all input of a certain length, and for all possible choices the machine can make. Their runtime is bounded by some polynomial independently of the choices they can make during a computation.

Definition 2 *A decision problem belongs to the complexity class $\mathcal{NP}$ if there exists a nondeterminisitic Turing machine running in polynomial time solving the problem.*

The second definition makes it easier to see that $\mathcal{P}$ is contained in $\mathcal{NP}$ because deterministic Turing machines are also NTMs. The question of whether the two classes are different or not is a famous open question in the theory of computation, commonly referred to as the $\mathcal{P} = \mathcal{NP}$ problem.

11.2 Complete Problems

12 Lambda-calculus

Alonzo Church developed a notation for arbitrary **functions** called λ-calculus. It is an extremely economical notation but at first sight somewhat cryptic, which stems from its origins in mathematical logic. Expressions in λ-calculus are written in strict prefix form. Further, function and argument are simply written next to each other without brackets around the argument.

12.1 Church-Rosser Theorem

Theorem 4 (Church-Rosser Theorem) *If a term M can be reduced (in several steps) to terms N and P, there there exists a term Q to which both N and P can be reduced (in several steps)*

This is sometimes known as confluence. Regardless of the order in which we decide to reduce our lambda calculus expression, we will always be able to arrive at the same term. This makes perfect sense - when evaluating an expression with arithmetic terms, we cannot get two different answers dependent on the order of evaluation. It follows that we get this corollary:

Corollary 1 *Every λ-term has at most one normal form.*

We can prove this. Assume there are two normal forms (for the case of contradiction) N and P to which are certain term M reduces. By the theorem of Church and Rosser there is a term Q to which both N and P can be reduced to. However, as we have assumed N and P are in their normal forms they don't allow any further reductions. The only possible interpretation for this is that N = P = Q.

12.2 Typed Lambda-calculus

Currently there is nothing in the grammar of lambda calculus that restricts us from forming awful terms. It would be completely possible to form $\sin \log$ at the moment - the sine function applied to the logarithm function. This is obviously impossible and any programming language would reject this as incorrectly formed. We are currently missing the notion of types, which represent what kind of arguements the function will accept and what it expects to return.

We can form the language for expressing these types very easily. We begin with base types such as `int` and `real`. On top of this we then form function types. This gives us the following grammar which is called the **system of simple types**:

$$\tau ::= c \mid \tau \to \tau$$

In this grammar, the value c is used as a placeholder for all of our base types we wish to include. Now we can form function types, we would give the type of the sine function

as `real` $\to$ `real`, which makes it obvious that it cannot accept the logarithm function as an argument. Using the type system that we have defined, we can create restrictions on what kind of terms are valid (or **well-typed**). We do this by an inductive definition:

- **Base Case**: For every type σ and every variable x, the term $x : \sigma$ is well typed and has type σ.
- **Function formation**: For every term M of type τ, every variable x, and every type σ, the term $\lambda x : \sigma.M$ is well-typed and has type $\sigma \to \tau$
- **Application**: If M is well-typed of type $\sigma \to \tau$ and N is well-typed of type σ then MN is well-typed and has type τ

Index