Resolution Machinery*

by Ramón Casares

The value of syntax is controversial: some see syntax as defining us as species, while for others it just facilitates communication. To assess syntax we investigate its relation to problem resolving. First we define a problem theory from first principles, and then we translate the theory concepts to mathematics, obtaining the requirements that every resolution machine has to implement. Such a resolution machine will be able to execute any possible resolution, that is, any possible way of taking a problem expression and computing the problem solutions. Two main requirements are found: 1) syntax is needed to express problems, that is, separate words are not enough, and 2) the resolution machine has to be as powerful as lambda calculus is, that is, it has to be Turing complete. Noting that every device that can generate any possible syntax, that is, any possible syntactically correct sentence of any possible, natural or artificial, language, has to be Turing complete, we conclude that syntax and problem resolving can use the same components, as, for example, sentences, functions, and conditionals. The implication to human evolution is that syntax and problem resolving should have co-evolved in humans towards Turing completeness.

Introduction

Problem, and other concepts in its semantic field, such as question, doubt, uncertainty, obstacle, difficulty, resolution, decision, plan, strategy, tactic, solution, satisfaction, target, aim, goal, purpose, answer, etc., are nearly everywhere. For example, when we talk about how something should be, we are describing a wished future state, but, at least implicitly, we are also judging the current state defective, and we are suggesting to take a path that reaches the target. Every time we look for some way to go from some state to some other better state we are in a problem resolving mood.

But, in spite of 'problem' pervasiveness, or, perhaps, because of it, and to the best of my knowledge, there is not any problem theory exact enough to be used in mathematics. So we will define one from first principles. The result is presented in section "Problem Theory". This theory uses just eight concepts: problem, freedom, condition, resolution, routine, trial, analogy, and solution. The theory will be evaluated later, so for now the reader should focus in getting an accurate view of each of the eight concepts. It is important to be aware of the distinction between 'solution' and 'resolution' that this theory makes: 'solution' is anything that satisfies the condition of the problem, while 'resolution' is the process of searching for solutions to the problem.

The next section, "Problem Resolving", is the core of the paper, but for anyone familiar with computing theory the bulk of the work was already done while presenting the problem theory. The aim of this section is to translate the eight concepts of the theory to mathematics. The first conclusion is that open expressions, also known as functions, are needed to translate problems to mathematics. And the last conclusion is that the whole power of Church's lambda calculus, that is, a Turing complete or a computationally universal device, is needed to execute any possible resolution. This last conclusion, that should not be surprising for computing theorists, should instead serve to demonstrate the value of the problem theory presented in the previous section.

In section "Syntax" we show, standing in the shoulders of Chomsky, that the whole power of a universal Turing machine, that is, a Turing complete or a computationally universal device, is needed to generate

* This is draft version `20140225`. Any comments on it to `papa@ramoncasares.com` are welcome.

any possible syntax, that is, any possible syntactically correct sentence of any possible, natural or artificial, language. This is another unsurprising conclusion, but, together with the previous one, we get something interesting: a syntax engine is a resolution machine, because both are universal computers.

So in the last section, "Implications", we explore some consequences beyond mathematics of the equation of syntax with problem resolution that we have found via Turing completeness. First we state that we are Turing complete, and then we conclude that syntax and problem resolving should have co-evolved in humans towards Turing completeness.

There is also an "Annex" with the answers to some frequently asked questions on problem theory.

Problem Theory

Before investigating what is needed to represent and to resolve problems, we should investigate what problems are; see [Casares 1993].

Problem

Every *problem* is made up of freedom and of a condition. There have to be possibilities and *freedom* to choose among them, because if there is only necessity and fatality, then there is neither a problem nor is there a decision to make. The different possible options could work, or not, as solutions to the problem, so that in every problem a certain *condition* that will determine if an option is valid or not as a solution to the problem must exist.

$$\text{Problem} \begin{cases} \text{Freedom} \\ \text{Condition} \end{cases}$$

Solution

A fundamental distinction that we must make is between the solution and the resolution of a problem. Resolving is to searching as solving is to finding, and please note that one can search for something that does not exist.

$$\begin{array}{r} \text{Resolving} \cdot \text{Searching} \\ \text{Solving} \cdot \text{Finding} \end{array}$$

Thus, *resolution* is the process that attempts to reach the solution to the problem, while the *solution* of the problem is a use of freedom that satisfies the condition. In the state-transition jargon: a problem is a state of uncertainty, a solution is a state of satisfaction, and a resolution is a transition from doubt to certainty.

$$\text{Problem} \longrightarrow \text{Resolution} \longrightarrow \text{Solution}$$

We can explain this with another analogy. The problem is defined by the tension that exists between two opposites: freedom, free from any limits, and the condition, which is pure limit. This tension is the cause of the resolution process. But once the condition is fulfilled and freedom is exhausted, the solution annihilates the problem. The resolution is, then, a process of annihilation that eliminates freedom as well as the condition of the problem, in order to produce the solution.

$$\underbrace{\left.\begin{array}{c} \text{Freedom} \\ \text{Condition} \end{array}\right\}}_{\text{Problem}} \xrightarrow{\quad\text{Resolution}\quad} \text{Solution}$$

A mathematical example may also be useful in order to distinguish resolution from solution. In a problem of arithmetical calculation, the solution is a number and the resolution is an algorithm such as the algorithm for division, for example.

Resolution

There are three ways to resolve a problem: routine, trial, and analogy.

$$\text{Resolution} \begin{cases} \text{Routine} \\ \text{Trial} \\ \text{Analogy} \end{cases}$$

To resolve a problem by *routine*, that is, by knowing or without reasoning, it is necessary to know the solution, and it is necessary to know that it solves that problem.

If the solution to a problem is not known, but it is known a set of possible solutions, then we can use a trial and error procedure, that is, we can try the possible solutions. To *try* is to test each possible solution until the set of possible solutions is exhausted or a halting condition is met. There are two tasks when we try: to test if a particular possibility satisfies the problem condition, and to govern the process determining the order of the tests and when to halt. There are several ways to govern the process, that is, there is some freedom in governing the trial, and so, if we also put a condition on it, for example a temporal milestone, then governing is a problem. And there are three ways to resolve a problem (*da capo*).

By *analogy* we mean to transform a problem into a different one, called question, which is usually composed of several subproblems. This works well if the subproblems are easier to resolve than the original problem. There are usually several ways to transform any problem (there is freedom), but only those transformations that result in questions that can be resolved are valid (which is a condition), so applying an analogy to a problem is a problem. There are three ways to resolve the analogy, the question, and each of its subproblems: routine, trial, and analogy (*da capo*). If we could translate a problem into an analogue question, and we could find a solution to that question, called answer, and we could perform the inverse translation on it, then we would have found a solution to the original problem.

$$\begin{array}{ccc} \text{Problem} & & \text{Solution} \\ \downarrow & & \uparrow \\ \text{Question} & \longrightarrow & \text{Answer} \end{array}$$

Eight Concepts

Lastly we are ready to list the eight concepts of the problem theory. They are: problem, with freedom and condition; resolution, with routine, trial, and analogy; and solution.

$$\begin{cases} \text{Problem} \begin{cases} \text{Freedom} \\ \text{Condition} \end{cases} \\ \\ \text{Resolution} \begin{cases} \text{Routine} \\ \text{Trial} \\ \text{Analogy} \end{cases} \\ \\ \text{Solution} \end{cases}$$

Problem Resolving

Once we know what the raw ingredients of problems are, we can now look for the means to execute any problem resolution; see [Casares 2010]. We will call every device that can execute any possible resolution a *resolution machine*. So in this section we will investigate the requirements that a resolution machine has to implement. We will collect the requirements by translating the concepts of the problem theory presented in the previous section to mathematics.

Unknown

To express a problem we have to refer to its freedom and to its condition. To name the freedom we have to use a word that does not refer to anything, that is, it has to be a word free of meaning. For example, if the problem is that we do not know what to do, then its more direct expression in English is 'what to do?'. In this sentence, the word 'what' does not refer to anything specifically, but it is a word free of meaning and purely syntactical.

In mathematics, the typical way to express the freedom of a problem is to use the unknown x, which works the same way as the interrogative pronoun 'what'. For example, if we want to know which number is the same when it is doubled as when it is squared, we would write:

$$x? \quad 2x = x^2.$$

The condition is, in this case, the equality $2x = x^2$. Equality is a valid condition because it can be satisfied, or not.

The x is just a way to refer to something unknown, so we could use any other expedient just by indicating it with the question mark (?). This means that

$$y? \quad 2y = y^2$$

is exactly the same problem. We call this equivalence α-conversion.

It is important to note that the unknown has to be part of the condition, in order to determine if a value is a solution to the problem, or not. In the condition, the unknown x is a free variable, and therefore the condition is an open expression, that is, a function.

In summary, in order to refer to the freedom of a problem we have to use free variables, which are words free of meaning that do not refer to anything. These free words are useless by themselves, we can even substitute one for another using an α-conversion, so they have to be combined with other words to compose the condition of the problem. We will call this structure of words a sentence. All things related to the sentence, as, for example, the rules for word combination, are what is usually called syntax.

Definition

We have just seen why we need syntax to represent a problem, and why separate words are not enough. We have also seen two types of word: semantic words with some meaning, and syntactical words without any meaning. Well, although it seems impossible, there is a third type of word: defined words.

A defined word is just an abbreviation, so we could go on without them, but they are handy. That way we substitute a word for a whole expression. We can, for example, define a word to refer to a problem:

$$q := \langle x? \;\; 2x = x^2 \rangle.$$

Ordered Pair

The resolution of the problem

$$x? \quad 2x = x^2$$

can go on this way:

$$\begin{aligned} 2x &= x^2 \\ 2x - 2x &= x^2 - 2x \\ 0 &= xx - 2x \\ 0 &= (x-2)x \\ [x-2=0] &\vee [x=0] \\ [x-2+2=2] &\vee [x=0] \\ [x=2] &\vee [x=0] \\ 2 &\vee 0\,. \end{aligned}$$

So it has two solutions, two and zero. In this case the resolution was achieved by analogy transforming the problem until we found two subproblems with a known solution, $\langle x?\ x=2\rangle$ and $\langle x?\ x=0\rangle$, which we could then solve by routine.

To represent each transformation the simplest expedient is the ordered pair, (s,t), where s and t represent two expressions: s before being transformed, and t once it has been transformed. Note that we can use a single ordered pair to express each transformation, or the whole sequence. For example, the complete sequence can be summarized in just one ordered pair, which, in this case, describes how to resolve the problem by routine, because s is the problem and t its solution:

$$\big(x?\ \ 2x = x^2,\ 2 \vee 0\big)\,.$$

Function

We can also resolve by trial and error. In the trial we have to test if a value satisfies the condition, or not, and the condition is an open expression. To note mathematically open expressions, also known as functions, we will use lambda expressions, $\lambda_x\xi$, where x is the free variable, and ξ the open expression. In the case of our problem:

$$\lambda_x[2x = x^2].$$

Now, to test a particular value a, we have to bind that value a to the free variable inside the condition. We also say that we apply value a to the function $\lambda_x\xi$. In any case we write the binding this way: $\lambda_x\xi(a)$. We can abbreviate the expression naming the function, for example $f := \lambda_x\xi$, to get the typical $f(a)$. In our case, 'to test if number 2 is equal when it is doubled to when it is squared' is written $\lambda_x[2x = x^2](2)$. And to calculate if a value satisfies the condition, we replace the free variable with the binding value; this process is called β-reduction. In our case we replace x with 2 ($x \leftarrow 2$), this way:

$$\lambda_x[2x = x^2](2) \to 2.2 = 2^2 \to 4 = 4 \to \textsc{yes}.$$

A condition can take only two values, which we will call YES and NO. In case we want to make a difference depending on the condition, and what else we could want, then we have to follow one path when the condition is satisfied (YES), and a distinct path when the condition is not satisfied (NO). To achieve this, a command **if** is used:

$$\textbf{if}\,\langle \textit{condition}\rangle\ \textbf{then}\,\langle \textit{case}\ \textsc{yes}\,\rangle\ \textbf{else}\,\langle \textit{case}\ \textsc{no}\rangle\,.$$

We can write down completely any trial just by using this two expedients: binding values to free variables in open expressions, and a conditional command, as **if**. Suppose, for example, that we guess that the solution to our problem is one of the first four numbers, in other words, that the solution is in set $\{1,2,3,4\}$, and that we want to try them in increasing order. Then we should nest four **if** commands:

$$\begin{array}{l} \textbf{if}\ \lambda_x[2x = x^2](1)\ \textbf{then}\ 1 \\ \textbf{else if}\ \lambda_x[2x = x^2](2)\ \textbf{then}\ 2 \\ \qquad \textbf{else if}\ \lambda_x[2x = x^2](3)\ \textbf{then}\ 3 \\ \qquad\qquad \textbf{else if}\ \lambda_x[2x = x^2](4)\ \textbf{then}\ 4 \\ \qquad\qquad\qquad \textbf{else}\ \textsc{no}\ . \end{array}$$

Condition

A known condition is a function with two possible outcomes, the loved YES and the hated NO. This means that each time we apply it we get a YES or a NO. For example:

$$\begin{aligned}
&\lambda_x[2x = x^2](0) \to \text{YES}\\
&\lambda_x[2x = x^2](1) \to \text{NO}\\
&\lambda_x[2x = x^2](2) \to \text{YES}\\
&\lambda_x[2x = x^2](3) \to \text{NO}\\
&\lambda_x[2x = x^2](4) \to \text{NO}\\
&\vdots
\end{aligned}$$

What resolves definitively a problem is the inverse function of the condition of the problem. Because the inverse function just reverses the condition, from a $0 \to$ YES to a YES $\to 0$, so when we apply YES to the inverse condition we get the solutions, and when we apply NO we get the set of the no-solutions. Thus, naming f the function $\lambda_x[2x = x^2]$:

$$\begin{aligned}
f^{-1}(\text{YES}) &\to \{0, 2\}\\
f^{-1}(\text{NO}) &\to \{1, 3, 4, \ldots\}.
\end{aligned}$$

We can use the condition in both directions: the natural direction, when we apply a value to test if it satisfies the condition, and the opposite direction, when we want to know what values satisfy the condition. To express a problem is enough to write the condition and to indicate which are its free variables, and to resolve a problem is enough to apply the condition in the opposite direction.

It is too easy to say that the resolution of a problem is the inverse function of its condition; I have just done it. Unfortunately, it is nearly always impossible to calculate that inverse function, and in some cases the condition is unknown. So, finding a resolution to a given problem is a problem.

Finding a resolution to a problem is a problem because it has its two mandatory ingredients. There is freedom, because there are several ways to resolve a problem, and there is a condition, because not every resolution is valid, but only those that could provide the solutions to the problem. And then, being a problem, we need a resolution to find a resolution to the problem. And, of course, to find a resolution to find a resolution to the problem we need, can you guess it?, another resolution. What do you think? Better than 'convoluted' say 'recursive'.

Tree

If we know the solution of a problem, then we can apply a routine, and write it as an ordered pair, as we have already done, $\big(x?\ 2x = x^2,\ 2 \vee 0\big)$. But we can also write it as a function, using the command **if**:

$$\lambda_\pi \left[\textbf{if } \pi \equiv \langle x?\ 2x = x^2\rangle \textbf{ then } 2 \vee 0 \textbf{ else } \text{NO}\right].$$

Here π is a free variable that we can bind to any expression, which is then compared ($\equiv$), not against the problem, but against the expression of the problem that we know how to resolve ($x?\ 2x = x^2$), and, **if** the comparison is successful, **then** we get its two solutions ($2 \vee 0$). When comparing open expressions, remember α-conversion and β-conversion.

I will not add anything new about trials, because we already know that we can express any trial just nesting command conditionals of bound functions. But remember that frequently determining the order of the tests is a problem.

By analogy we transform a problem in another problem, or problems. Most times the outcome will be more than one problem, because 'divide and conquer' is usually a good strategy for complex problems. So, in general, the resolution to a problem will be a tree, being the original problem its trunk. If we use analogy

to resolve it and we get, for example, four easier subproblems, then the tree has four main branches. But, again, from each branch we can use analogy, and then we get some sub-branches, or we can use routine or trial. We resolve by routine when we know the solution, so the subproblem is solved; these are the leaves of the resolution tree. Trial ...

You got lost? Don't worry, even myself get lost in this recursive tree, and to what follows the only important thing to keep in mind is one easy and true conclusion: expressions that represent resolutions to problems have a tree structure, because they describe the resolution tree. You also has to know that it is enough to allow that the elements of a ordered pair be ordered pairs to be able to build binary trees with ordered pairs.

Recursion

Resolution is a process that, when it is successful, transforms a problem into a solution. But this neat and tidy view hides a more complex resolution tree, one that is full of subproblems. Inside the tree we see partial resolutions, which are atomic transformations transforming a subproblem into a subproblem.

The solution to a problem can also be a problem or a resolution. For example, when a teacher is looking for a question to include in an exam, her solution is a problem. And when an engineer is designing an algorithm to implement some general type of electrical circuits, her solution is a resolution. Note that, in this last case, a previous step could be a sub-optimal algorithm to be enhanced, and then one of the transformations would take a resolution to return a resolution; this also happens with solutions.

Any open condition is a problem, if we are interested in the values that satisfy the condition. An open condition is an open expression, and closed expressions are just open expressions with no free variables, that is, with zero free variables. In addition, you can close any open expression by binding all its free variables. So, with this in mind, we can treat open expressions as expressions. Not every open expression is an open condition, but only generalizing to full functions we can cope with the resolution to resolution transformation that the engineer above needed.

The conclusion is then that a partial resolution can start with any expression and can end with any expression. So a resolution has to be able to transform any expression into any expression.

$$\left.\begin{matrix}\text{Problem}\\ \text{Resolution}\\ \text{Solution}\end{matrix}\right\} \xrightarrow{\text{Resolution}} \left\{\begin{matrix}\text{Problem}\\ \text{Resolution}\\ \text{Solution}\end{matrix}\right.$$

Please note that the resolution can be the transformation, but also what is to be transformed, and what has been transformed into. We call this feature of transformations that can act on themselves recursivity.

Quoting

Recursivity needs a quoting expedient to indicate if an expression is referring to a transformation, or it is just an expression to be transformed.

Joke:
— Tell me your name.
— Your name.
Who is answering has understood "tell me 'your name'", which could be the case, although unlikely. In any case, quotation prevents the confusion.

Quoting is very powerful. Suppose that I write:

En cuanto llegué, el inglés me dijo: "I was waiting you to tell you something. ...

$$\vdots$$

[a three hundred pages story in English]

$$\vdots$$

... And even if you don't believe me, it is true". Y se fue, sin dejarme decirle ni una sola palabra.

Technically, I would have written a book in Spanish.

Technically, using the quoting mechanism, all symbolic languages are one. As we learned from Chomsky, there is only one language; see [Chomsky 2000]. And, as in the Spanish book, even if you don't believe me, it is true.

Requirements

We are ready to summarize the capacities necessary to execute any possible problem resolution. In other words, these are the requirements that every resolution machine, lets call it $\mathcal{R}$, has to implement.

$\mathcal{R}$ uses three types of words.

- Semantic words: These are the link to semantics.
- Syntactic words: These are the core of $\mathcal{R}$, so in this section we will note its name, between parenthesis, in Lisp dialect Scheme; see [Abelson and Sussman 1985].
- Defined words: These are just handy abbreviations.

$\mathcal{R}$ needs, then, a dictionary to keep the abbreviations, and operations to add a dictionary entry defining an abbreviation (`define`), and also to modify or to delete an entry (`set!`).

The words are the atoms that can be used to compose sentences. A sentence has a tree structure, so inside a sentence we can find other sentences. Both words and sentences are called expressions. Because there are two kind of expressions, words and sentences, $\mathcal{R}$ needs a condition to test the type of an expression (`atom?`). $\mathcal{R}$ needs operations both to compose sentences from expressions (`cons`), and to take the expressions from the sentences where they are in (`car`, `cdr`).

$\mathcal{R}$ also needs a condition to check if two expressions are equal (`equal?`). Most times, this depends eventually on semantic equality, but, in any case, $\mathcal{R}$ has to work its part, for example honoring α and β-conversions on open expressions.

$\mathcal{R}$ sentences can be open expressions (`lambda`), also known as functions. These functions are recursive, meaning that any expression can be applied to the function using β-reduction, or, in other words, that any expression can be bound to any of the free variables of the open expression. Because of this feature, functions can be applied on functions, and a quoting expedient is needed to indicate that the expression is not referring to a function (`quote`).

Finally, $\mathcal{R}$ needs a conditional command so its behaviour can depend on the result of a condition (`cond`).

Resolution Machine

The requirements in the previous subsection define a limited Lisp that, nevertheless, is a superset of Church's lambda calculus; as defined in [Church 1935]. As Turing proved that lambda calculus is computationally universal, see [Turing 1936], the conclusion is that to execute any possible resolution a Turing complete device is required. Or saying the same thing in other words, a resolution machine is a universal computer.

$$\text{Resolution Machine} = \text{Universal Computer}$$

This shows that the problem theory presented is complete, because adding more concepts to it would not result in a more capable device.

Three Limits

Having seen that a resolution machine is a universal computer, we must conclude that the limits of resolution machines are the same as those of universal computers. Only the terminology differs. Please skip this subsection if you are not interested in the relation between problem theory and computing theory.

In problem theory if a problem has solutions, then we say that it is *solvable*, and if it has not any solution, then we say that it is unsolvable. On the other hand, all problems are resolvable in the sense that, for example, we can apply a trial and error resolution to any problem. So we can reserve 'resolvable' to more specific problems. For example, it can be proved that $\langle x?\ 2x = x^2 \wedge x > 2\rangle$ has no solutions, and in this case we will say that this problem, which is unsolvable, was resolved, and therefore it is unsolvable but resolvable.

So, there is a resolvable problem that is not solvable, and we have found the *solution limit* of resolution machines, or Turing complete devices in problem resolvers: a resolution machine can execute any resolution, but it cannot solve some problems.

A famous unsolvable but resolvable problem is the halting problem. Firstly see that a resolution can never halt. This is the case, for example, if we apply a trial resolution, testing each integer number from zero up until we get a solution, to any unsolvable problem. So it would be nice to know, for any pair of resolution and problem, if the resolution will halt on the problem, or not. The halting problem condition would be satisfied by any resolution that would always halt, and that would say, for any pair (resolution $\mathcal{R}$, problem $\mathcal{P}$), if the resolution $\mathcal{R}$ will halt on the problem $\mathcal{P}$, or not. Note that the solution of the halting problem should be a resolution, but, unfortunately, the halting problem has no solution; as proved in [Turing 1936].

Now we can define precisely what we mean when we say that a problem is resolvable. A problem is *resolvable* when there is a resolution to the problem that finds all the problem solutions, and then halts; otherwise the problem would be unresolvable. See that, for a resolvable problem with an infinite number of solutions, we are not asking the resolution to halt, because that would not be possible. So resolvable in problem theory is equivalent to recursively enumerable in computing theory.

In computing theory a set is recursive if its characteristic function is computable. In other words, a set of strings is recursive if there is a Turing machine that always halts, and that correctly tells whether any finite string belongs to the set, or not. Therefore, if the set of the solutions to a problem is a recursive set, then the condition of the problem is computable. In that case, every Turing complete device can express the problem, and therefore we say that the problem is *expressible*. So recursive in computing theory is equivalent to expressible in problem theory.

A famous theorem in computing theory says: there is a recursively enumerable set that is not recursive; see [Arbib 1987]. It is translated to problem theory as: there is a resolvable problem that is not expressible. We can now write the *problem limit* of resolution machines, or Turing complete devices in problem resolvers: a resolution machine can execute any resolution, but it cannot express some problems.

In subsection "Condition", we have seen that finding a resolution to a problem is a problem, and now that there is a resolvable problem that is not expressible. Then finding a resolution to an inexpressible but resolvable problem, lets call it $\mathcal{P}$, is a solvable but unresolvable problem, lets call it $\mathcal{Q}$. Problem $\mathcal{Q}$ is solvable because problem $\mathcal{P}$ is resolvable. Problem $\mathcal{Q}$ is unresolvable because the algorithm finding the resolutions cannot take problem $\mathcal{P}$ expression as argument, and yet it would have to return the valid resolutions to problem $\mathcal{P}$. This shows that there is a solvable problem that is not resolvable, and we can write the *resolution limit* of resolution machines, or Turing complete devices in problem resolvers: a resolution machine can execute any resolution, but it cannot resolve some problems.

In summary, resolution machines have limitations on each of the three main concepts of the problem theory: a resolution machine can execute any resolution, but it cannot express some problems (problem limit), and it cannot resolve some problems (resolution limit), and it cannot solve some problems (solution limit).

$$\text{Problem} \longrightarrow \text{Resolution} \longrightarrow \text{Solution}$$

Syntax

Grammar

Chomsky presented, in [Chomsky 1959], a hierarchy of grammars. A *grammar* of a language is a device that is capable of enumerating all the language sentences. And, in this context, *language* is the (usually infinite) set of all the valid sentences.

At the end of SECTION 2 in that paper, we read: "A type 0 grammar (language) is one that is unrestricted. Type 0 grammars are essentially Turing machines". At the beginning of SECTION 3, we find two theorems.

THEOREM 1. For both grammars and languages, type 0 $\supseteq$ type 1 $\supseteq$ type 2 $\supseteq$ type 3.

THEOREM 2. Every recursively enumerable set of strings is a type 0 language (and conversely).

Then THEOREM 2 is explained: "That is, a grammar of type 0 is a device with the generative power of a Turing machine."

From the two theorems we can deduce three corollaries:

COROLLARY 1. The set of all type 0 grammars (languages) is equal to the set of all grammars (languages).
This is because, according to THEOREM 1, type 0 is the superset of all grammars (languages).

COROLLARY 2. For each Turing machine there is a type 0 grammar (and conversely).
This is equivalent to THEOREM 2, but in terms of grammars (devices) instead of languages (sets).

COROLLARY 3. For each Turing machine there is a grammar (and conversely).
This results by applying COROLLARY 1 to COROLLARY 2.

Syntax Engine

We will call every device that can generate any possible language a *syntax engine*. For each Turing machine there is a grammar and for each grammar there is a Turing machine, see COROLLARY 3, and therefore a syntax engine has to be able to behave as any possible Turing machine, that is, it has to be Turing complete. In other words, to generate any possible language a computationally universal device is required. Or more concisely: a syntax engine is a universal computer.

$$\text{Syntax Engine} = \text{Universal Computer}$$

We saw that a resolution machine is a universal computer, and that a universal computer is a syntax engine, so a resolution machine is a syntax engine.

$$\text{Resolution Machine} = \text{Universal Computer} = \text{Syntax Engine}$$

Syntax Definition

We have chosen 'syntax engine' rather than 'language engine' to name a device that can generate any possible 'language' because 'language' in [Chomsky 1959] refers only to 'syntax'. It does not refer to semantics, because meanings are not considered, nor to pragmatics, because intentions are not considered.

So now we will provide the definition of syntax that follows from [Chomsky 1959] after adjusting for this deviation towards language: *syntax* consists of transformations of strings of symbols, irrespective of the symbols meanings, but according to a finite set of well defined rules; so well defined as a Turing Machine is. This definition of syntax is very general and includes natural languages syntax, just replacing symbols with words and strings with sentences, but it also includes natural languages morphology or phonology, taking then subunits of words or sounds.

As an aside, please note that this definition of syntax manifests, perhaps better than other definitions, what we will call the syntax purpose paradox: syntax, being literally meaningless, should be purposeless. It is easy to underestimate the value of some mechanical transformations of strings of symbols.

Implications

Finiteness

In this section we will investigate some implications outside mathematics of what we have seen until now. And outside mathematics we meet finite entities, so we need to be aware of some additional limitations.

Finite computational devices have two additional limitations: finite memory and finite time. So the results of computing theory should always be completed with a cautious 'provided they have enough time and memory' when they are applied to finite devices, though the provision is usually taken for granted.

See an example that we will use later. When Turing wrote his paper, in 1936, a computer was a person. So the Turing machine, as it was presented in [Turing 1936], models the calculations done by a human computer. This means that we are Turing complete provided we have enough time and memory.

This time we have explicitly expressed the provision, but in what follows we will omit it, in the understanding that the provision is tacitly stated when referring to finite entities.

Problems and Evolution

We are Turing complete, but not all living individuals are Turing complete, so we might wonder what difference does this make. A more limited individual will apply its limited set of resolutions to a problem, but a Turing complete individual can try new resolutions without any limit. The key point is that a single Turing complete individual can, in principle, try any possible resolution, any possible way to resolve a problem, while an individual that is not Turing complete can only try those resolutions that are implemented in the hardware of its body, mainly in the hardware of its brain.

Species that are not Turing complete need a genetic change to modify its set of resolutions, while Turing complete individuals can try new resolutions without any hardware change, but just by a software change. So the timing of creativity depends on evolution until Turing completeness is achieved, and it does not depend on evolution after that point for the species that achieve Turing completeness. We will call every point where an evolutionary path achieves Turing completeness an *evolutionary singularity*.

New behaviours related to the resolution of problems, such as feeding, mating, and, in general, surviving, should proliferate after an evolutionary singularity. And noting that a tool is the physical realization of a resolution, then an explosion of new tools should start whenever a evolutionary singularity happens. So the archaeological record of human tools should point to our evolutionary singularity. In summary, creativity is slow until an evolutionary singularity, and creativity explodes after every evolutionary singularity.

Nevertheless, Turing completeness is defined by a single condition: pan-computability. A universal Turing machine, which is the prototype of Turing completeness, is every Turing machine that can compute whatever any Turing machine can compute. This means that to run a specific computation you can use either the specific Turing machine that runs that specific computation or a universal Turing machine. In other words, creativity is the only exclusive behavior of Turing complete problem resolvers. And this explains an elusive fact: every time a specific behavior is presented as uniquely human, it is later rejected when it is found in another species. The point is not that we behave in some specific way to resolve a problem, but that we are free to try any behavior we seem fit to resolve our problems. Creativity is the mark of Turing completeness.

Syntax and Problems

We could equate problem resolution with syntax, because problem expression is not possible without syntax, but mainly because the requirement that full problem resolution impose is universal computability, which is the maximum requirement. So full problem resolution needs syntax, and needs all of it: full problem resolution needs the whole syntax.

By equating syntax to problem resolution we have solved the syntax purpose paradox: when a brain become a syntax engine, then that brain become a resolution machine. So syntax is meaningless but very useful and its purpose is to resolve problems in any possible way.

Syntax and problem resolution converge at the end, but not at the beginning. This means that a full recursive syntax engine is equal to a full resolution machine, but also that a minimal syntax has little, if any, relation to problem resolution. For example, the first step of syntax could be very simple: just put the agent before every other word. This simple expedient by itself would prevent efficiently some ambiguities, explaining who did and who was done something.

Syntax recursivity is credited with making possible the infinite use of finite means, which is obviously true. But our wanderings in the problem resolution realm have shown us that there are other features provided by recursive syntax that are, at least, as important as infinity; infinity that, in any case, we cannot reach. My own three favorites are: 1) sentences, or hierarchical tree structures; 2) functions, or open expressions with free variables; and 3) conditionals, or expressing possibilities. *I cannot imagine how we would see the world if there were no conditionals, can you?*

Syntax and Evolution

Because full syntax is the same as full problem resolving, the evolution of syntax and the evolution of problem resolving both lead to the same destination. Syntax and problem resolving should have co-evolved towards Turing completeness.

I would argue that one of the very first steps, the sentence, was driven by language, that is, by syntax. The reason is that ambiguity is reduced efficiently just by imposing some order to a set of words, and this is the case even without free variables, which was possibly the first reason why problem resolving needed sentences.

I would argue that the last step, fulfilling universal computability, was driven by problem resolving. The reason, now, is that Turing completeness is useful for problem resolving to detach itself from hardware causing an explosion of creativity, but it is not so useful to syntax and language.

I don't know which one, syntax or problem resolving, was driving each of the other steps that provided each of the capabilities that are needed to implement a Turing complete brain, but my feeling is that the bulk of the task was done by language. If this was the case, then language was driving human evolution to Turing completeness, not to achieve a full resolution machine, which was a side effect, a "collateral damage", but because syntax features, such as sentences, open expressions, and conditionals, made communication more efficient resulting in more fitted individuals.

In any case, it would be difficult to deny that syntax was, at least, instrumental in achieving Turing completeness, and therefore, that syntax was influential in reaching our evolutionary singularity.

Beyond the Singularity

These results should help us to reevaluate syntax. In language evolution there are two main positions regarding syntax; see [Kenneally 2007]. The continuist side defends a more gradual evolution, where syntax is just a little step forward that prevents some ambiguities and makes a more fluid communication; see [Bickerton 2009]. For the other side, led by Chomsky, syntax is a hiatus that separates our own species from the rest; see [Hauser, Chomsky, and Fitch 2002]. What position should we take?

The co-evolution and convergence until fusion of syntax and problem resolution explains that, once our ancestors achieved Turing completeness, they acquired full syntax, and their brains became resolution machines. Then they could see a different the world. How much different? Very much.

Seeing the world as a problem to resolve implies that we look for the causes that are hidden behind the apparent phenomena. But it is more than that. Being able to calculate resolutions inside the brain is thinking about plans, goals, purposes, intentions, doubts, possibilities. This is more than foreseeing the future, it is building the future to our own will. Talking about building, what about tools? A tool is the physical realization of a resolution, so, with full syntax we can design tools, and even design tools to design tools.

Thinking the world as a problem to resolve implies also that we calculate how to use freedom in order to achieve our objectives. So, yes, we are free because of syntax; see more on this in [Casares 2012].

Summarizing, the identification of syntax with problem resolution explains why syntax, being so little thing, has made us so different. And, although syntax is more than just one operation, so we have probably acquired the whole syntax in more than one evolutionary step, my vote goes, anyway, to ... Chomsky!

Conclusion

Our species is Turing complete. Therefore we must explain the evolution of Turing completeness.

Turing complete individuals can transform strings of symbols, irrespective of the symbols meanings, but according to any possible finite set of well-defined rules. It seems a nice capability but, being meaningless, not very useful. It could be, but syntax is also about meaningless transformations of strings of symbols according to rules. Turing completeness is then a pan-syntactical capability, because the syntax of a natural language does not need to follow any possible set of rules, but only one specific set of rules.

Syntax is very peculiar indeed, because syntax is a peculiarity of human language, which is our most peculiar faculty. For this reason, we can argue that syntax is what defines our species, and yet it seems improbable to explain how some mechanical transformations of strings of symbols have made us like we are. In addition, syntax is not our only peculiarity: is it a coincidence that we are the only species with a syntactical language and also the most creative?

But then, after realizing that Turing completeness is closely related to syntax, which is a human peculiarity, we have to halt, because we cannot progress anymore without new inputs. In order to overcome that impasse, this paper introduces a new piece of the jigsaw puzzle that links areas that were previously unconnected: the new piece is problem resolving. Problem resolving is a piece of problem that fits with syntax on one side, and with Turing completeness on the other side.

We are now ready to understand Turing completeness from the perspective of problem resolving. After finding that creativity is the peculiarity of Turing complete problem resolvers, we see how our Turing complete problem resolving machinery explains both our creativity and our pan-syntactical capability. But before that, we have also seen that Turing complete problem resolving needs sentences, functions, and conditionals; all of them employed by syntax.

The conclusion is that syntax and problem resolving should have co-evolved in humans towards Turing completeness.

Annex: Frequently Asked Questions on Problem Theory

These are the answers to some frequently asked questions on problem theory, which are included in order to prevent some misunderstandings.

A resolution machine is not a resolver

A resolution machine is a universal computer, and this means that it is Turing complete, as it is, for example, a universal Turing machine. Then, to compute the problem solutions, the resolution machine shall take a problem expression and a resolution expression. So a resolution machine is not a whole resolver, but a necessary part of every universal resolver. Every universal, or Turing complete, resolver needs, in addition to the resolution machine, some machinery to identify and express the problem that the resolver is facing, and some machinery to select and express the resolution to execute. And I call every universal resolver with a body to actually apply the solutions that are found a subject, but that is another story.

That 'a resolution machine can execute any resolution' does not imply that 'a resolution machine will resolve any problem'

We have seen in the main body of the paper that there are unresolvable problems. Of course, a resolution machine cannot resolve unresolvable problems, but even in the case of problems that are expressible, solvable, and resolvable, finding a solution is not assured.

A valid, or successful, resolution is any resolution that resolves a problem, and this means that a successful resolution returns all of the problem solutions, and then halts. Some resolutions are partially successful,

because they find some solutions, but not all of them. Some other resolutions are just wrong, because what they return as solution does not satisfy the problem condition, although, some times, this is because the resolution is returning approximate solutions. So "any resolution" includes both successful and unsuccessful resolutions, even the plain wrong ones.

The conclusion is that the universal resolver machinery that selects the resolution faces the problem of choosing a successful resolution out of an infinite set of mostly wrong resolutions.

That 'full syntax is needed to execute any resolution' does not imply that 'syntax, or full syntax, is needed to resolve problems'

The relation between syntax and problem resolving is wide, deep, and long. It starts in the beginnings, because syntax is needed just to express problems, and it goes till the very end, because a full syntax engine is equal to a full resolution machine; in the end syntax equals problem resolving. In between, syntax and problem resolving share features as sentences, functions, and conditionals. But, we have already seen this.

Now suppose that whenever some kind of bacteria needs more light to get more energy, it senses where it receives the largest intensity of light, and it moves in that direction. From the problem theory point of view, the bacteria is resolving its energy problem applying a known solution, that is, applying a routine. The routine is implemented chemically, and it does not transform strings of symbols. So it is false that syntax, or full syntax, is needed to execute problem resolutions, because to execute simple resolutions, as, for example, routine resolutions, a Turing complete device is not needed, not at all. Only when the possibility of executing any resolution is required, Turing completeness is needed. And 'full syntax' is synonymous with 'Turing completeness'.

What is so good about Turing completeness?

A Turing complete device is a general purpose computer. For example, a personal computer (PC) is Turing complete, while an arithmetical calculator is not Turing complete. This implies that the arithmetical calculator will only resolve those problems that are one arithmetical operation away. On the other hand, the personal computer (PC) can, in principle, execute any possible resolution, that is, any possible program, but you will have to get, or write, the program and load it into the personal computer (PC) just to perform one arithmetical operation. In addition, for any possible program, we can build a dedicated computing device that will behave as the program running in the general purpose computer, except that it will run the calculations more efficiently. So what is so good about Turing completeness?

Broadly speaking, Turing completeness transforms computing devices into data. And building computing devices costs much more than feeding data, because building hardware needs huge amounts of time and energy, while writing software just needs some time and a little amount of energy, and copying software needs even less.

In the case of evolution, as we saw, Turing completeness detaches problem resolving from evolution, causing an explosion of creativity. Creativity explodes because, after achieving Turing completeness, performing new ways of resolving the survival problems becomes cheap, easy, and available to single individuals, while, before achieving it, any single new way of resolving them required genetic changes on species over evolutionary time spams.

Acknowledgments

I am grateful to Ray Jackendoff, Jim Hurford, Diego Krivochen, and Marc Hauser, for their comments on previous versions of this paper.

References

[Abelson and Sussman 1985] Harold Abelson, Gerald Sussman, Julie Sussman, *Structure and Interpretation of Computer Programs*; The MIT Press, Cambridge MA, 1985, ISBN: 978-0-262-01077-1.

[Arbib 1987] Michael Arbib, *Brains, Machines, and Mathematics*, Second Edition; Springer-Verlag, New York, 1987, ISBN: 978-0-387-96539-0. It is Theorem 6.3.9 in page 140, which is an abstract form of Gödel's incompleteness theorem (page 165).

[Bickerton 2009] Derek Bickerton, *Adam's Tongue: How Humans Made Language, How Language Made Humans*; Hill and Wang, New York, 2009, ISBN: 978-0-8090-1647-1. See paragraph between pages 235 and 236.

[Casares 1993] Ramón Casares, *Ingeniería del aprendizaje: Hacia una teoría formal y fundamental del aprendizaje*; Universidad Politécnica de Madrid, 1993 (in Spanish). This reference points to the doctoral dissertation where this problem theory was first presented; it was later published in [Casares 1999].

[Casares 1999] Ramón Casares, *El problema aparente: Una teoría del conocimiento*; Visor Dis., Madrid, 1999, ISBN: 978-84-7774-877-9 (in Spanish).

[Casares 2010] Ramón Casares, *El doble compresor: La teoría de la información*; `www.ramoncasares.com`, 2010, ISBN: 978-1-4536-0915-6 (in Spanish). Section "Problem Resolution" in this paper is adapted from sections §78 to §95 of this reference.

[Casares 2012] Ramón Casares, *On Freedom: The Theory of Subjectivity*; `www.ramoncasares.com`, 2012, ISBN: 978-1-4752-8739-4.

[Chomsky 1959] Noam Chomsky, "On Certain Formal Properties of Grammars"; in *Information and Control*, Volume 2, Issue 2, pp. 137-167, June 1959.

[Chomsky 2000] Noam Chomsky, *New Horizons in the Study of Language and Mind*; Cambridge University Press, Cambridge, 2000, ISBN: 978-0-521-65822-5. See page 7.

[Church 1935] Alonzo Church, "An Unsolvable Problem of Elementary Number Theory"; in *American Journal of Mathematics*, Vol. 58, No. 2 (Apr., 1936), pp. 345-363. Presented to the American Mathematical Society, April 19, 1935.

[Hauser, Chomsky, and Fitch 2002] Marc Hauser, Noam Chomsky, Tecumseh Fitch, "The Language Faculty: Who Has It, What Is It, and How Did It Evolved?"; in *Science* 298, pp. 1569-1579, 2002.

[Kenneally 2007] Christine Kenneally, *The First Word: The Search for the Origins of Language*; Penguin Books, New York, 2008, ISBN: 978-0-14-311374-4. See Chapter 15.

[Turing 1936] Alan Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem"; in *Proceedings of the London Mathematical Society*, Volume s2-42, Issue 1, pp 230-265, 1937. Received 28 May, 1936. Read 12 November, 1936.