Refactoring of the Clause Database

Manuel Bonarrigo

July 11, 2020

Contents

1 Introduction

It comes a moment in the lifetime of a software product development when the envisioned domain has been completely transferred into a well engineered design: the core functionalities have been abstracted into strongly cohesive, loosely coupled, interfaces; patterns have been diffused through the system to be resilient to any form of change, will it come from domain, topology, or libraries; new features are integrated seamlessly; client code start to flourish, and people seems to be pretty happy about writing it.
That is the moment developers wipe their foreheads, for good results have been achieved.
That is the moment optimization starts.

The following work approaches the recurring problem in Prolog implementations of storing the multitude of clauses that constitutes the program knowledge base. The implementation at stake is the porting of the tuProlog project [1] to a multi-platform framework completely rewritten in Kotlin, at supporting both the JVM and JavaScript platforms[2]. There are two simple key concepts behind the storage of clauses, especially when the theory is used in a dynamic fashion to be interacted with programmatically: *a)* relative order matters, since Prolog semantic can only be respected if the clauses are resoned about in the same order they were written and *b)* it needs to be as fast as possible, since it will be massively accessed by the Prolog solver engine in order to achieve the expected program output. The previous implementation, while being perfectly fine from a design standpoint, lacked to achieve the first point. At the moment of inspecting the database content, variables would end up clustered as the initial component of every result. This strife with the desire of programmatically interact with the clauses, since losing a common sense of the insertion order meant losing the ability to write a Prolog program, hindering the language correctness.

Spotting the source of this problematic behaviour has meant following the path of a clause since the moment it is created, to the moment it would actually be stored into the data structure, to the moment it is retrieved according to a rightful query. This eventually lead to a complete refactoring of the data structure, to both address problematics encountered or unlock strategic opportunities recognized along the path.

What the refactoring mainly addressed was the loosely coupling of the clause access interface (referred as the `ClauseDatabase` from now on) from the actual clause storage facility (referred as `ReteTree` from now on) through the creation of a collection framework that would abstract away the control over the `ReteTree` and allow for the reuse of the clause storage utilities it implements to fulfill other purposes.

As the refactoring was being carried out, it emerged the possibility of introducing the first parameter indexing feature, allowing for a faster retrieval where the proper set of clauses are asserted into the theory, and the proper set of query is used to retrieve them. This led to another refactoring, completely

centered on the `ReteTree`, changing the ways it stores clauses and adding means to optimize exploration performances wherever it could be done.

2 State of the art

2.1 Rete algorithm overview

The Rete algorithm is the backbone of many rule-based expert systems. In fact, it is an efficient solution for managing dynamic knowlodge bases storing both data tuples (e.g. Prolog facts) and production rules (e.g. Prolog rules). In particular, Rete supports the storage of a large amount of facts and rules (generally referred as clauses, henceforth) in such a way that:

- information insertion is order-sensitive, meaning that the order in which clauses are insertet affects the way they are retrieved or removed;
- information retrieval occurs by pattern matching (e.g. via logic unification) and its computational cost – in terms of time – does not increase as the amount of clauses in the knowlodge base increases;
- clauses may be both dynamically removed and inserted efficiently, in an ordered way

It is worth to be mentioned that the original formulation of Rete also takes into account the possibily of executing rules, and the side effects that their execution may have on the knowledge-base—in terms of insertion or removal. Within the scope of this project, we do not consider the latter aspect because, in 2P-Kt, the resolution and storage modules are strictly separated. Thus, the focus of this project is on storage alone, there including information insertion, retrieval, and removal.

The main idea behind Rete – and also the one we take inspiration from – is that clauses should be recursively indexed on a per-type basis, in order to group similar clauses, making the computational cost of information retrieval independent from the total amount of clauses. Using the Prolog nomenclature, this means that each clause is decomposed into its structural parts – namely, the *head* and the *body* –, and each part is possibly structurally decomoposed again. This decomposition is recursively performed over and over again until a non-deconposable item is met—e.g. a constant or a variable. Thus, the decomposition produces tree in which each node groups (i.e. indexes) all the clauses sharing all the features from the root node to the current node—similarly to what happens for radix trees.

Let us provide an example. Consider for instance the clause $f(a)$:-b. It can be decomposed into the $f(a)$ and b parts, representing the clause head and body respectively. Then, each part can be further decomposed again—even if, in the particular case of Prolog, only the heads of the clauses are relevant for indexing. For instance, the head is composed by a functor (f), an arity (1), and an argument (a), which is indeed an atom. Accordingly, the Rete tree storing that clause should have

1. a node for the root (grouping all clauses in the knowledge base)

2. a node for all the clauses having both a head and a body (grouping all rules)

3. a node for all rules whose functor is f

4. a node for all rules whose functor is f and whose arity is 1

5. a node for all rules whose functor is f, whose arity is 1, and whose first argument is a

and the aforementioned clause should then be stored as a child of node 5. Adding novel clauses may imply novel nodes to be properly added to this hierarchy, at the most adequate level.

Summarising, in the reminder of this document, we describe an implementation of the Rete algoritm tailored on the needs of Prolog and the 2P-Kt system. For this reason, the proposed solution differs from the original proposal of the algorithm in a number of way. Nevertheless, the structural-decomposition-based approach of Rete has heavily inspired the *core* of our algorithm and for this reason we retained its name.

While the Rete network algorithn is not implemented by this means into the system built, the structural decompositional approach of the α nodes has heavily inspired the *core* of the algorithm.

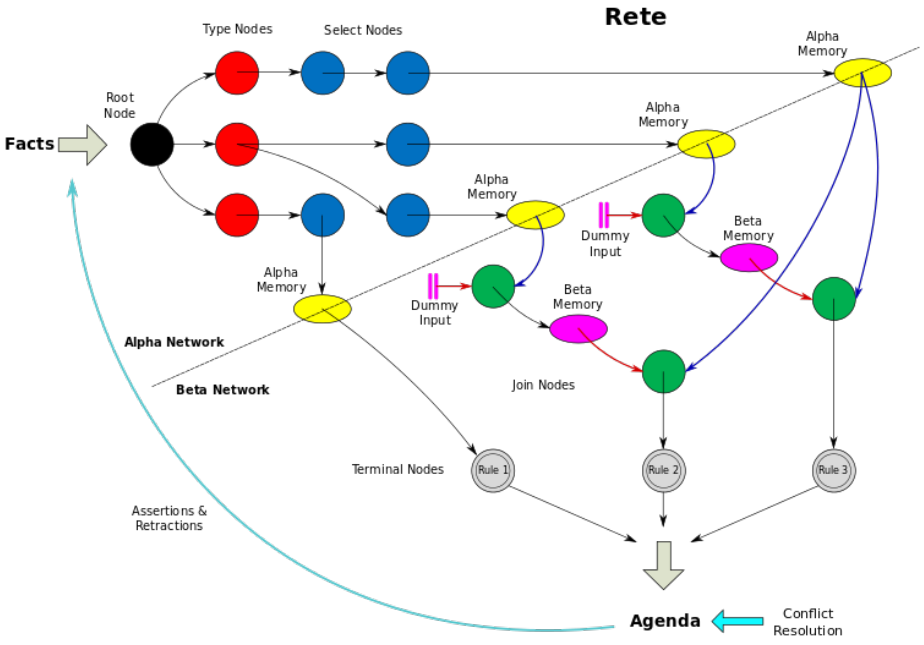

Figure 1: A detailed overview of the Rete network (source Wikipedia)

3 Design

3.1 ClauseDatabase

The `ClauseDatabase` (fig. 2) is the conceptual *façade* to uniformly access the theory of a Prolog runtime. It consists of an immutable data structure providing the necessary methods to operate upon the declared clauses. After the refactoring, it can be instantiated in two fashions, to better adapt to any needs the client code could have: the *ordered* and the *unordered* one. The order difference hinted at is with regards to the sequence of clauses offered at instantiation time of the data structure, and their order relation with the other members of the clause family [1].

In the *ordered* implementation this order is guaranteed to be always respected, so that it becomes possible to write Prolog programs through the concatenation of cascading clauses; in the unordered implementation, since no reordering algorithm needs to be triggered, what it is obtained is a boost in the performance of the read and retrieve operations, namely *get* and *retract*. This becomes useful at the moment of using a Prolog theory for its data storing capabilities.

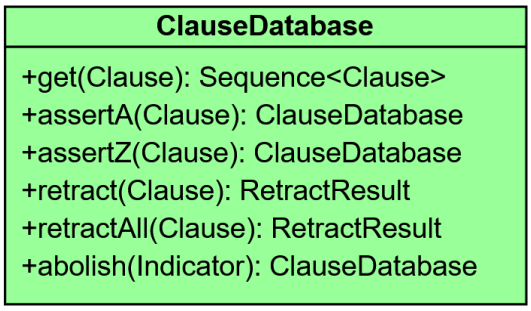

Figure 2: The interface of the `ClauseDatabase`, reduced to the *core* concepts

3.2 Collection framework

The collection framework is a facility added with regards to the needs of `ClauseDatabase`, so to loosen the coupling between the high-abstraction-level of managing the theory and the low-level data structure actually managing all the `Clause`. What it is abstracted away is the control over the of *ordered* and *unordered* theory, so that at implementation level one could simply switch collection to achieve the desired behaviour. Moreover, having an interface in place,

[1] With *family* it is intended what is commonly called *indicator*, that is having the same functor and arity. Since `Indicator` is already declared in the p2Kt type hierarchy, *family* will be used when talking about the concept, *indicator* when referring to the type in the context of the code

a new kind of collection can be implemented at any moment, so to shape any type of behaviour desired to be followed by the `ClauseDatabase`.

The typologies of collection implemented at this very moment are (in both the mutable and immutable flavour) the `ClauseQueue` and the `ClauseMultiSet`, with the final objective of providing the form of control aforementioned upon the relative ordering of the clauses.

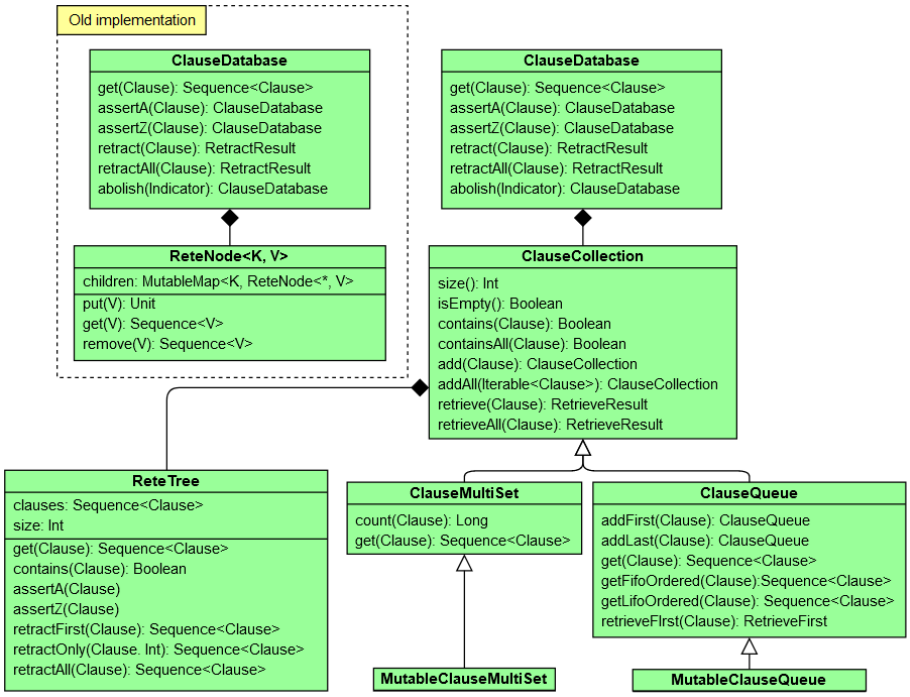

Figure 3: A high level class diagram depicting the refactoring of the `ClauseDatabase`.

There is not much surprise in the functionalities offered by these data structures. They have been realized by taking as example the classical collection of the Java Framework. For this reason, the nomenclature chosen for the methods does not retrace the Prolog naming, but the Java Collection Framework one instead. The reason stands in the *least astonishment principle*, applied to future developers that put themselves in the shoes of writing client code: the meaning of *collection* has too many implications in terms of mnemonic usage. For this reason, it is present the `add()` method, instead of the `assert()` method, and `retrieve()`) instead of `retract()`. Differently from the Java Collection Framework, the retrieve method actually returns some kind of reference to the removed object, instead of a `boolean`. This is due to the absence of the `equality` concept, since the rightful clauses to be retracted are found through *unification*, which means that the input `Clause` is not guaranteed to be the same removed

from the data structure.

Something that is worth to be noted is that the collection framework is not designed with a generic purpose in mind. Its functionality is heavily linked with the Rete algorithm, which is in turn heavily linked to the structure of the set of logical rules in use. Being so strictly bound to this kind of domain constraints does not leave space, nor does require, that this kind of data structure should be able to perform its operation on any other kind of class other than `Clause`. There is obviously room for improvements, as discussed in section 7.

3.3 ReteTree

Following the footprints of the Rete algorithm, the new implementation of the `ReteTree` takes various steps to partition and categorize the `Clause` offered to be asserted or retracted, so that it can incrementally delegate the appropriate operation to the correct member of the tree. The high level structure and functionality of each node will be presented along the example asserting the `Clause` $[f(a)$:-_$]$ inside the theory of the tree. To this purpose, unaware of the actual signatures of the nodes, let's assume that every node is capable of performing an assertion of a `Clause` over its own knowledge base. The following descriptions can be appreciated graphically in fig. 4.

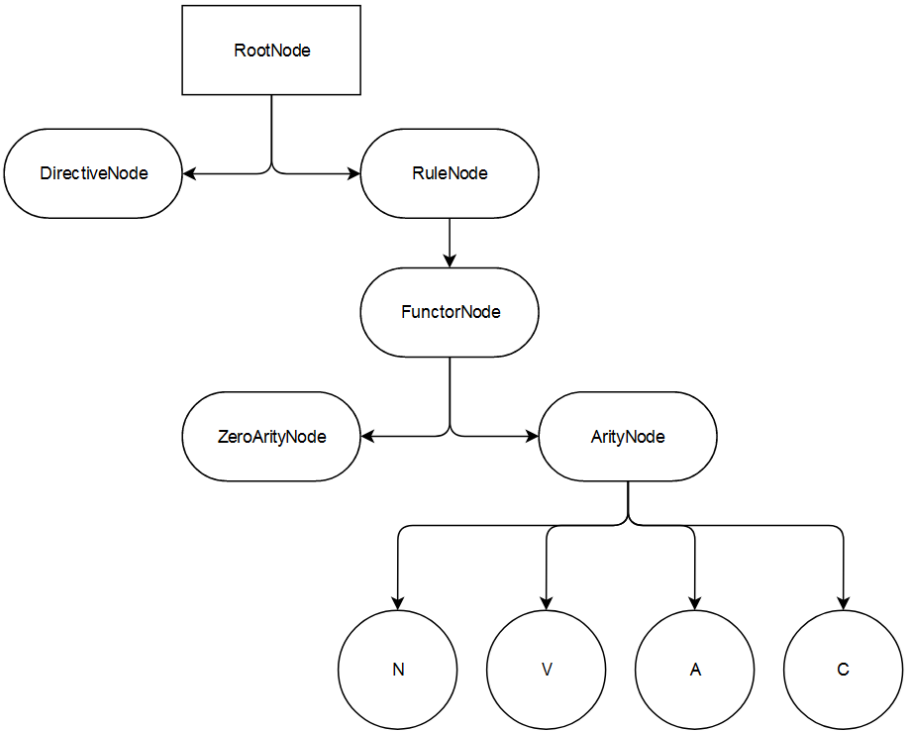

Figure 4: An abstract depiction of how the `ReteTree` will be represented at memory level. N, V, A, and C, represent the actual nodes where the clause storage happens, indexing them for their first argument

RootNode As the highest level of branching, in the `RootNode` it is made the first skimming of the `Clause`: it is discerned if it has a head or not, directioning the flow to the `DirectiveNode` if the property is missing, or delegating to the `RuleNode` if it is present. Moreover, the `RootNode` is the place where a `Clause` is transformed into an `IndexedClause`, a data structure that decorates the original clause associating it to an unique numeric identifier: from now on, this data structure will be the carrier of the information to be stored. A proper description, and all the insights behind the powerfulness of such a simple concept can be found in section 4.4. For the comprehension of the following part, `Clause`, `IndexedClause` and the not-yet-introduced `SituatedIndexedClause` can be treated as if it does not exist any difference between the classes. Still, in the text will be used the appropriated form for the level of the tree under current analysis.

DirectiveIndex The `DirectiveIndex` is absolutely straightforward. Here are stored all clauses without a head, and that's pretty much it. Whenever the `RootNode` detects an operation to be performed on a `IndexedClause` that

is recognized as a `Directive`, this node is the one in charge of providing an answer. The administration of the clauses it contains is the same of another index node, as `VariableIndex`, and for this same reason the explanation of the storage mechanism of an *index* will be postponed to after the rest of the tree has been completely described.

RuleNode The first real step into the Rete algorithm takes place into the `RuleNode`. At this level the head of what is now an `IndexedClause` is analyzed, to understand from which `Functor` it is described by. Once discovered, a `FunctorNode` for the specific functor is created: from now on, all the `IndexedClause` described by that same functor will be delegated to this new node, and will be the responsibility of the `RuleNode` to pass the operations requested to the correct `FunctorNode`. Following the example we were asserting into the `ReteTree`, f is detected as the functor of the head, and a `FunctorNode` relative to all future f are created. The whole `IndexedClause` is passed to the new node, so that it can be continue to walk towards the index leaf nodes.

For the sake of conciseness and despite its impropriety, from now on it will be omitted that the analysis on the `IndexedClause` actually happens to its head. Moreover, the body is never looked at in the entirety of the theory submodule, so that any confusion would be impossible

FunctorNode Once in the `FunctorNode`, the arity of the `IndexedClause` is analyzed. In our example it would be calculated as 1. An `ArityNode` relative to all the clauses described by the functor f and arity of 1 will be created, and the `IndexedClause` will be forwarded to this recipient.

ArityNode The `ArityNode` is the place where it begins the analysis over the arguments of the clause, so that the actual indexing can be performed. There are only four possible outcomes for the type of the first argument of a clause: it might only be an `Atom`, a `Numeric`, a `Variable`, or a `Compound`. They will be respectively stored in an `AtomIndex`, a `NumericIndex`, a `VariableIndex`, or a `CompoundIndex`. Any `ArityNode` comes with all these type of indexex already instantiated. In our example, the first argument would result in being an `Atom`, so the `IndexedClause` is forwarded to such an index.[2]

AtomicIndex The `AtomIndex`, along with most of the other leaf nodes, takes a very simple role. It associates the first argument of the `IndexedClause` to the whole clause itself, creating a dictionary that maps every `Atom` encountered which is described by a functor f, with arity 1 (in the case of our example).

There is another perk that is achieved by reaching a leaf index node: the `IndexedClause` becomes a `SituatedIndexedClause`, which means establishing a bound with the index the clause settled in. This bound gives a future caller the

[2]The functionalities that the `ArityNode` brings to the table are many more, and will be discussed in depth when talking about the retraction of clauses

ability to ask the index for the deletion of the specific clause the caller possesses a reference to. This choice will make sense soon.

NumericIndex There is no particular difference between an `AtomIndex` and a `NumericIndex`, with obvious exception of the type of the first argument that led to this leaf.

VariableIndex A `VariableIndex` slightly differs from the `AtomIndex` and `NumericIndex` by the means that having a `Variable` as a first argument does not provide anough peculiarity to create a dictionary: there is not that much difference between $[f(X)\text{:-\_}]$ and $[f(Y)\text{:-\_}]$. This implies that there aren't actually means to create a dictionary. The `SituatedIndexedClauses` here are just stored sequentially.[3]

ZeroArityNode A `ZeroArityNode` is another sequential indexer. The head of the clauses stored here are pure functors, and again, it is not present any information that could distinguish a clause from another.

CompoundIndex Last, but not the least, and actually not even the last, the `CompoundIndex`. The main role of this index is to create the `SituatedIndexed-Clause` which first argument is identified as a *compound term*. And provide the means to efficiently store and retrieve them, indexing the whole clause recursively by the first argument of the first argument of the actual `IndexedClause`. Immediately switching to an example, let's consider the clause $[f(g(1))\text{:-\_}]$. It is described by the functor f, with arity of 1, so it does follow the same path of our previous example, $[f(a)\text{:-\_}]$. At the `ArityNode`, its first argument has been detected as a compound, delegating the management to the `CompoundIndex`. The whole rationale of the tree is just applied again to the *inner* compound: the `CompoundIndex` we're in at this very moment takes the role of a `RuleNode`, and only $g(1)$ is analyzed.

[3] This is the same behaviour of the `DirectiveIndex`, as there is no clue in the head of a `Directive` hinting at any difference between them.

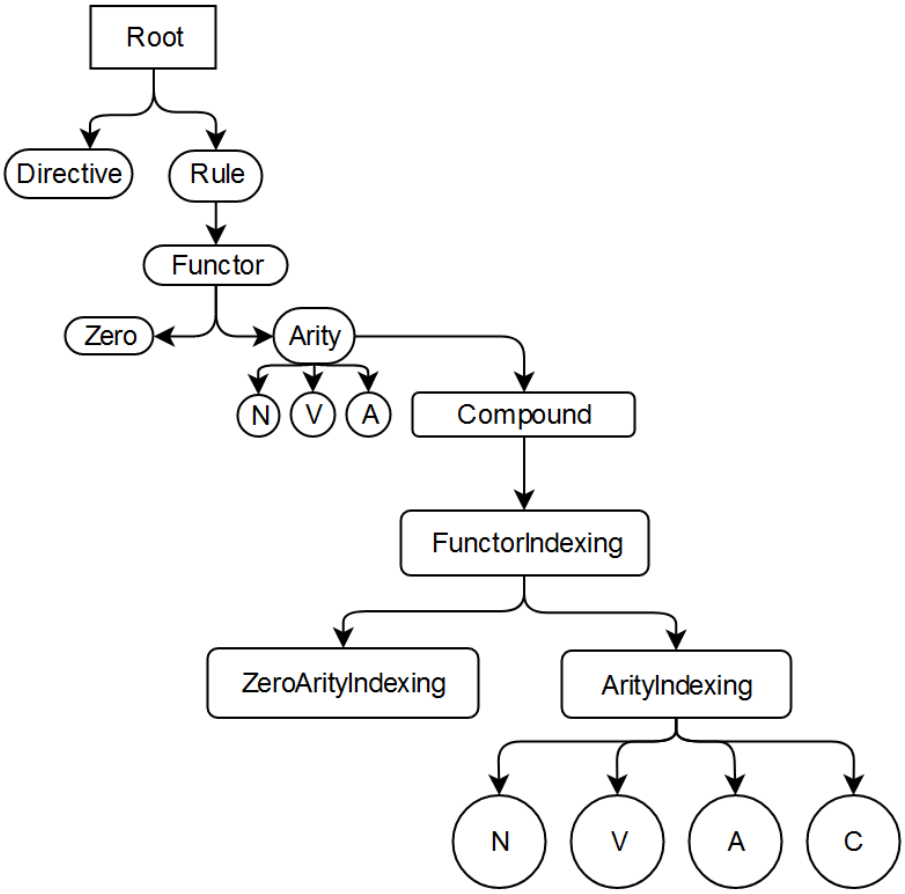

Figure 5: The recursive structure of the ReteTree, repeated for every time a new level of inner compound is found during the analysis of a `Clause`

At this point of the explanation it should be trivial that a `FunctorIndexingNode` will be created and associated to the existence of g, followed by the creation of an `ArityIndexingNode` regarding the arity of g, 1, and that given its first argument is a `Numeric`, the `IndexedClause` will be *situated* by a `NumericIndexing`. What is not trivial is the presence of two different implementations to describe the existence of the functor and arity, but the motivations for this will be further explained in section 3.3.4.

Situating an `IndexedClause` in a `CompoundIndex` marks the end of the digression over the assertion process. Before switching to the retraction, it is necessary to give more details upon the concrete structure of the `ReteTree`, so that some design choices, thoughts and consequences could be anticipated in order to simplify the clutter of information that the retraction chapter will be

nonetheless.

3.3.1 Clause family

The decoration of a `Clause` is at the very foundation of the system.

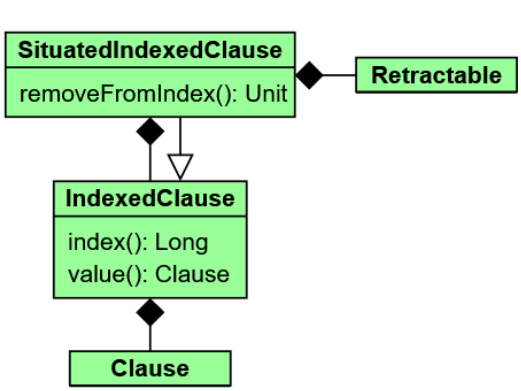

Figure 6: The class diagram of `Clause` decorations at the foundation of the whole system

The creation of an `IndexedClause` happens as soon as the `Clause` is delivered to the `RootNode` for assertion. It is paired with a unique, incremental, identifier based on the type of the assertion: for assertZ it would be created a positive identifier bigger than the one used for the last recorded tail assertion, for assertA it would be created a negative identifier smaller than the one used for the last head assertion. Through this simple mechanism it is possible to recreate the original assertion order, even if the clauses have been scattered along the whole tree.

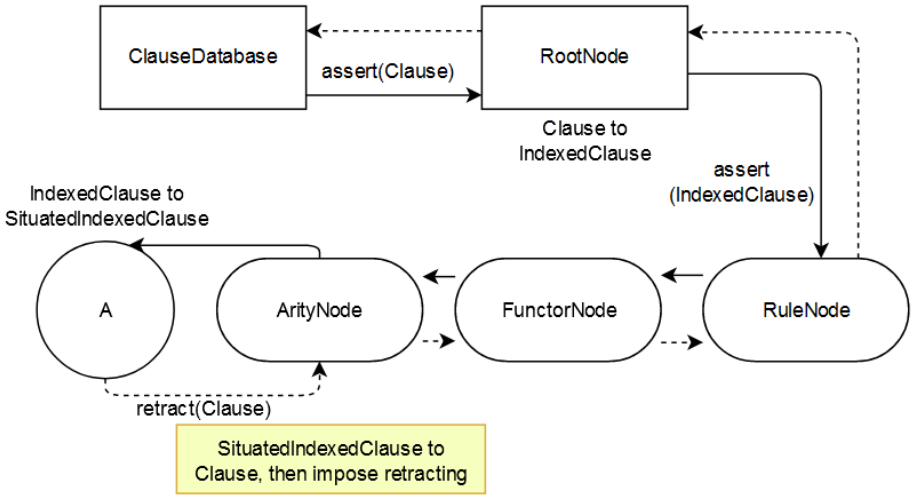

Figure 7: The transformation path a `Clause` endures at runtime

The second decoration pairs an `IndexedClause` with a reference of a `Retractable` object. To the extent of this paragraph, let's just identify that as any node where a `SituatedIndexedClause` can be retracted from. Owning such a reference gives the clause the peculiar characteristic of being able of removing itself from the index it is stored in. The reason behind this capability is the possibility that in front of some particular kind of retraction, the node that would choose if the clause is actually going to get retracted is a different node from the one phisically holding the clause. This problem will be analyzed more accurately in section 4.4

3.3.2 ReteTree design

To have a more concrete vision of how the structure in fig. 5 and to provide a better naming for the concepts that are going to be encountered from now, let's switch to a UML diagram, expressing the effective relations between classes.

While the main behavioural characteristics of all the implementation classes have been already clarified (with the exception of the `ArityNode`, at section 4.4, `ArityIndexingNode` and `FunctorIndexingNode`, both on section 3.3.4) the interfaces giving meaning to all the declared nodes are still missing.

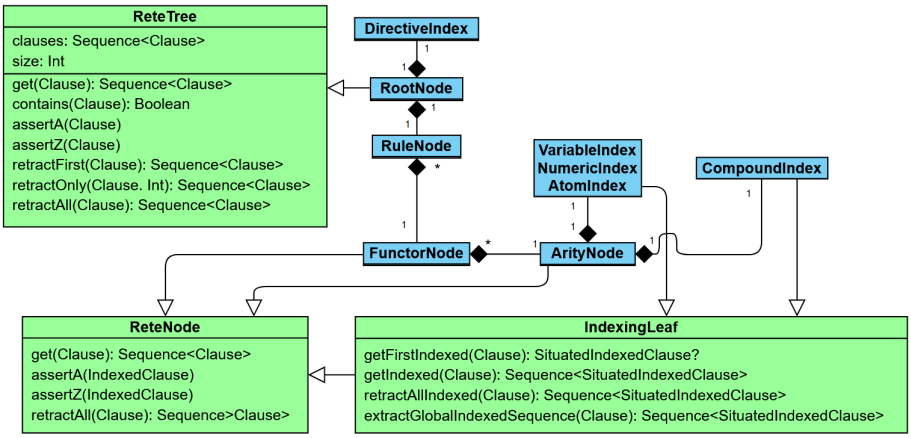

Figure 8: The core class diagram of the system

Running top to bottom through the tree again, the first met interface is `ReteTree`. It is the only public interface, enabling the whole data structure to be used from client code. To this extent it is worth noting that `RootNode`, its only implementor, has nothing in common with the other nodes in the tree. This is because the responsibility of this first layer is to offer functionalities directly to clients. By isolating the root from the other nodes, its behaviour and appearance is not influenced by any kind of implementation related changes that would occur in the rest of the tree, so that it can continue its role of published interface, and all the nodes in the subtree can freely change structure and behaviour.

The following are the functionalities a `ReteTree` has to offer. This sloppy list of methods will be done just here, since most of the methods are going to be delegated through the tree until reaching the leaves. For every method returning a multitude of clauses, the natural order will be imposed based on the nature of the tree itself. The methods are:

1. **get(Clause) -** all the clauses that match the given one are collected and given in output, *lazily*. After this method call, all the clauses obtained are still in the theory;

2. **contains(Clause) -** tells if there exist any clause that will unify over the given one.

3. **assertA(Clause) -** if the tree is ordered, this method will assert the given clause at the beginning of its own index. It will behave as `assertZ(Clause)` otherwise.

4. **assertZ(Clause) -** the given clause will be asserted at the end of the index it is designed to fall into.

5. **retractFirst(Clause) -** the clause recognized as the first one, according to the ordering rationale of the tree, is spotted, removed and returned.

6. **retractOnly(Clause, Int) -** the same as `retractFirst(Clause)`, but limited to a certain amount of clauses instead that only one.

7. **retractAll(Clause) -** retracts all the clauses matching the given clauses, returning them.

Most of these methods can be seen declared as well by the `ReteNode` interface. Whoever extends such a node is actually in charge of delegating or providing a direct output for this method, if it is able to, e.g. the `IndexingLeaf` interface. Its implementors might be able to directly provide an answer to inquiried clauses, but it may also be possible that their result would just be needed to be further analyzed by some upper layer node in ordder to provide the rightful answer to an inquiry. Briefly, this is the meaning of the methods:

1. **getFirstIndexedClause(Clause) -** it is returned the nullable `SituatedIndexedClause` that this particular index sees as its own first.

2. **getIndexed(Clause) -** all the clauses matching the given one are returned by the natural order the particular `IndexingLeaf` has imposed on them.

3. **retractAllIndexed(Clause) -** all the clauses that would match the given one are retracted from this index, and then returned. The difference between this method and the `ReteNode` one is that in this case a group of `SituatedIndexedClause` is returned, since it is needed by an upper layer to choose the correct ordering to propose the output

4. **extractGlobalIndexedClause(Clause) -** sometimes it is detected a query `Clause` that would simply refers to *all* the content of a particular index, e.g. $f(X)$:-Y over the f/1 subtree. In this case, every check is canceled[4], and the whole content is returned as a result.

3.3.3 TopLevelNode and Retractable

The `TopLevel` and `Retractable` interfaces represent two capabilities that, respectively, only some kind of `ReteNode` and `IndexingLeaf` can have. They are nonetheless strictly related.

Being a `TopLevelNode` means having the rightful capability of finely-grained choosing which `SituatedIndexedClause` has to be retracted when several options are available. It is not a matter of *unification*. It is a matter of being so high on the tree's layers to be fully aware of what the correct order of the clauses is. `TopLevelNode`s are then the only ones able to declare what does *first* mean.[5] But ironically, they can't do nothing about it. They have received several clause references, but don't have any idea where the data structures actually holding the clauses are. On the contrary, a `Retractable` has exactly that type of knowledge, and that's exactly what it is meant to do. At this point it should be clear

[4] hence it is named as *global*

[5] Again, the section 4.4 will completely explain all the problems related

why every clause has a reference to the `Retractable` index where it is *situated*. Doing so, it does not matter how many tree levels there are from the actual index holding a clause and the node actually able to choose about its fate: as soon as a `SituatedIndexedClause` has been chosen for retraction, the `Clause` is is holding is extracted to be later returned, and the `SituatedIndexedClause` is imposed to delete itself. Its inner `Retractable` reference is then asked to remove from its own data structures the very reference of *this* (from the clause's point of view). And the clause is gone.

An alternative would have been scanning the tree again through the clause-s arguments so to find again the correct index, which is just wasted computational power. Having the `Retractable` reference embedded into the clause means that the computational cost of its actual retraction is only bound to the one needed by the index.

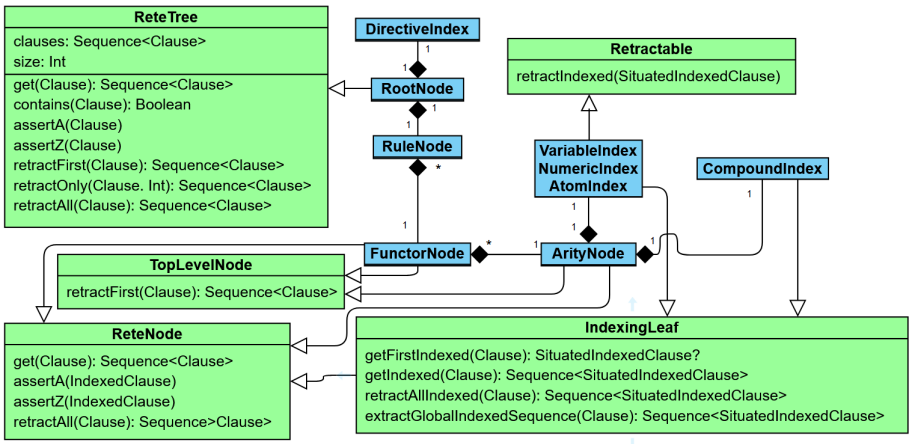

Figure 9: A more complete version of the system

3.3.4 IndexingNode

To understand where does the distance between a `TopLevelNode` and a `Retractable` come from, it is necessary to unveil the most intricate part of the system (when looked through UML): the `IndexingNode` hierarchy.

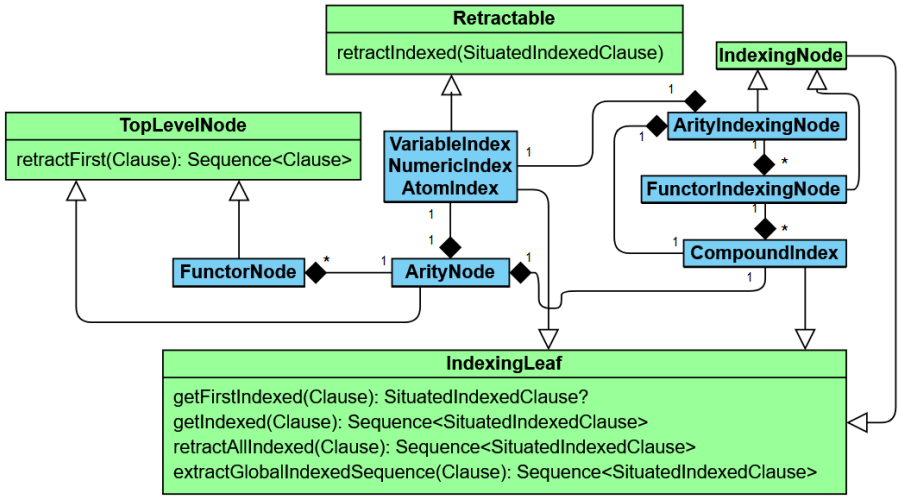

Figure 10: A snapshot of the IndexingLeaf hierarchy

By just giving a small glance to fig. 13, the whole structure becomes immediately more clear. An `IndexingNode` can only exist as a sub-tree for a `CompoundIndex`[6]. This is because if there isn't any *compound recursion* the present indexes are already able to store any possible clause.

The difference between a *top-level* ArityNode (resp. FunctorNode) and an ArityIndexingNode (resp. FunctorIndexingNode) resides in the missing strength upon their own decision when it come to the retractability of the `SituatedIndexedClause` they are asked to deliberate about. They can inquiry their own sub-trees for a result, but there 's nothing more they can do but pass it upwards, until a `TopLevelNode` able to choose is reached.

A slightly more subtle difference comes from the selector that an `IndexingNode` can receive. Let's tackle this through an example. Consider $f(g(h))$ and $f(m(n))$ as heads of two different clauses; they both belong to the f/1 *top-level* branch. Both their first argument are classified as `Compound`, so the control is passed over to the `CompoundIndex` in charge of managing f/1 terms. Their first argument is then rolled out; the former comes from the g/1 family, the latter from m/1. Two different `FunctorIndexingNode` with two subsequent `ArityIndexingNode` are then created, each one managing one of the clauses, which become *situated* in the respective `AtomIndex`.

Then it comes the moment a `get` operation is invoked. The `Clause` inquiried is $f(X)$:-Y. The f/1 part of the clause is easy to navigate, as a properly named node does exist for both properties, but then it comes the turn for the `CompoundIndex` to proceed with the rest of the extraction, as no `Retractable` node has been hit yet. X has to be matched, which means whatever functor

[6]Something appreciable by the last diagrams is that a `CompoundIndex` is not a `Retractable`, since it does not contains any clause

branch, on whatever arity branch. This is the fundamental difference between a `TopLevelNode` and an `IndexingNode`: X cannot exist as a query at a *top-level*, because it is illegal to start a query with a variable. The code could have never been the same, and so a different set of classes were born.

3.3.5 Refactoring Overview

The refactoring of the `ReteTree` led to a design explicitly oriented towards computational efficiency. Every aspect of the problem has been partitioned into its own class so that every bit of optimization can be introduced at a fine grained level as soon it is felt the necessity to do so.

The structure of the tree is obviously still very similar to the old one. The biggest decompositional steps are inspired by the Rete algorithm, and behaved well even before the refactoring. The problems arised as soon as the actual storage of the `Clause` took place, leading to an inconsistent theory that would make impossible to operate programmatically. This was obviously the point most of the changes were made.

Since the refactoring was carried on a partially working codebase, there wasn't a clear direction to follow. The first step was to design an algorithm that would empower the tree to perform a reordering of the clauses given as input without caring about the place these would be stored. As a mandatory parallel job, the tree structure was sketched. The ideas forging the algorithm and the tree structure eventually joined, so that the structure just presented emerged without solution of continuity. Most of the work was done around the `IndexingLeaf` hierarchy, as it was the main source of problematic behaviours.

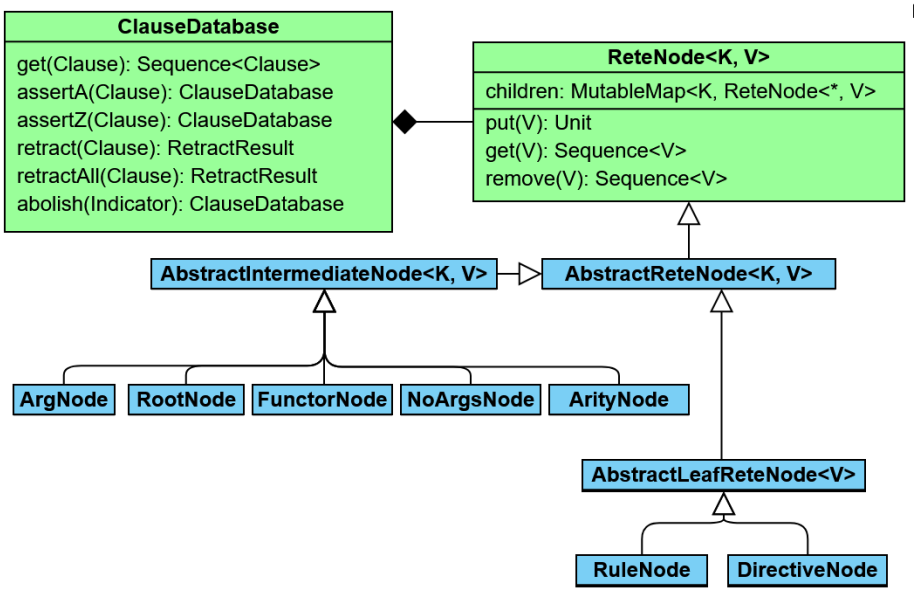

Figure 11: The class diagram of the previous implementation of the ReteTree

One of the biggest difference between the previous (in fig. 11) and the actual implementation (in fig. 12) is the complete removal of any kind of generic class from the tree codebase. This choice follows the one taken for the collection framework, but there is also a technological reason behind it. Trying to avoid diving too much into technicalities[7], *erasure* and a complex taxonomy of concepts reified into a strongly typed hierarchy does not go along very well. This becomes extremely true when a different behaviour is expected to be triggered accordingly to the particular instance of concept one is manipulating at the moment. Once the erasure process is allowed into the system, obtaining this kind of information becomes a cumbersome process that not only shadows the solution space of the original problem, but also cripples the development with a swamp of boilerplate only needed to tamper the complexity of the process itself. What is being blathered about is *reflection*, and as powerful as it is, it is not a mechanism that should be preferred to a clean node hierarchy if not as a last resource to cope with inherited technical debt. [3]

[7] And saltiness. I'm not a big fan of erasure

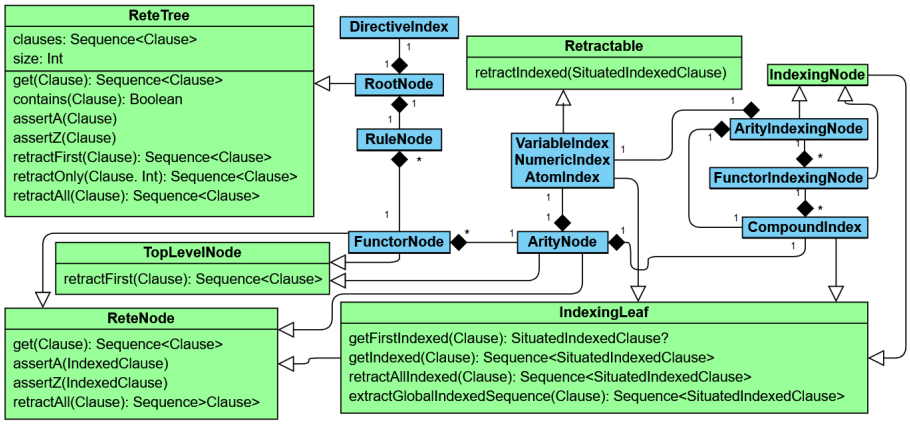

Figure 12: The complete class diagram of the actual implementation of the ReteTree

4 Implementation

Before dwelling into the implementation related perks, let's consolidate every explanation given until now while also paving the road for the future ones with a running example. In fig. 13 it is presented an instantiated `ReteTree`, obtained through the commands at table 1.[8]

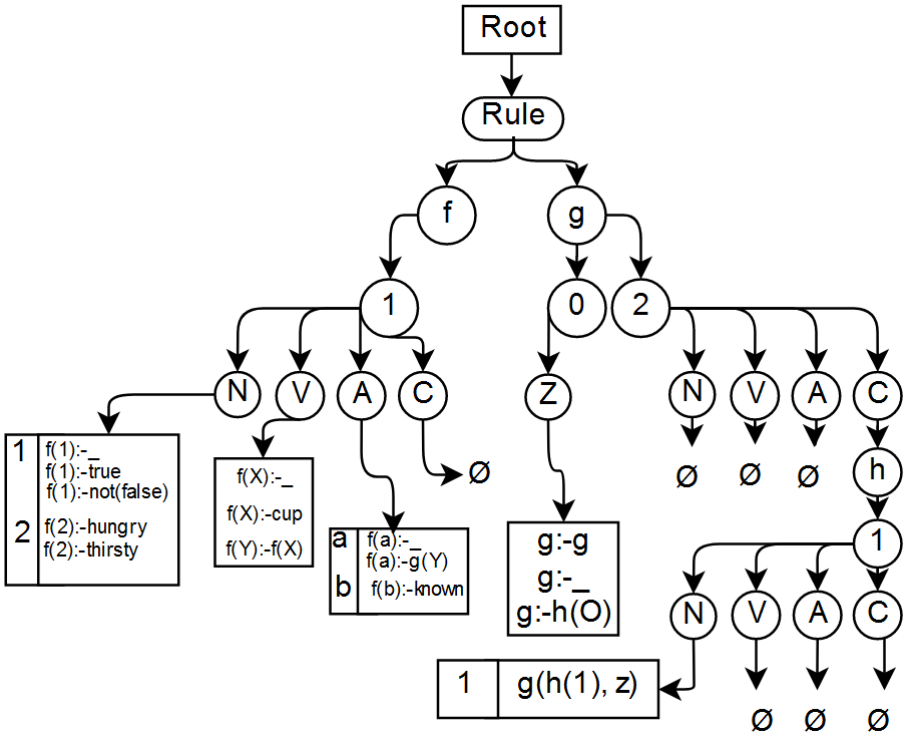

Figure 13: A running example of a `ReteTree`

[8]The columns of the table are meant to be read as three parallel timeline. Rows are not correlated.

Input commands	Indexing	Output theory
`assertA(`$f(Y)$`:-`$f(X)$`)`	$\langle$-7, $f(1)$:-_$\rangle$	g:-g
`assertZ(`$f(2)$`:-`$thirsty$`)`	$\langle$5, $f(1)$:-$true\rangle$	g:-_
`assertZ(`$f(b)$`:-`$known$`)`	$\langle$6, $f(1)$:-$not(false)\rangle$	$f(1)$:-_
`assertA(`$f(X)$`:-`cup`)`	$\langle$-3, $f(2)$:-$hungry\rangle$	$f(X)$:-_
`assertZ(`$f(a)$`:-`$g(Y)$`)`	$\langle$1, $f(2)$:-$thirsty\rangle$	$f(a)$:-_
`assertA(`$f(2)$`:-`$hungry$`)`	$\langle$-6, $f(X)$:-_$\rangle$	g:-$h(O)$
`assertA(`g`:-`$h(O)$`)`	$\langle$-2, $f(X)$:-$cup\rangle$	$f(2)$:-$hungry$
`assertA(`$f(a)$`:-_)`	$\langle$-1, $f(Y)$:-$f(X)\rangle$	$f(X)$:-cup
`assertZ(`$g(h(1), z)$`:-_)`	$\langle$-5, $f(a)$:-_$\rangle$	$f(Y)$:-$f(X)$
`assertZ(`$f(1)$`:-`$true$`)`	$\langle$3, $f(a)$:-$g(Y)\rangle$	$f(2)$:-$thirsty$
`assertZ(`$f(1)$`:-`$not(false)$`)`	$\langle$2, $f(b)$:-$known\rangle$	$f(b)$:-$known$
`assertA(`$f(X)$`:-_)`	$\langle$-9, g:-$g\rangle$	$f(a)$:-$g(Y)$
`assertA(`$f(1)$`:-_)`	$\langle$-8, g:-_$\rangle$	$g(h(1), z)$:-_
`assertA(`g`:-_)`	$\langle$-4, g:-$h(O)\rangle$	$f(1)$:-$true$
`assertA(`g`:-`g`)`	$\langle$4, $g(h(1), z)$:-_$\rangle$	$f(1)$:-$not(false)$

Table 1: A depiction of the three states of a theory computation in the built system. To the left it's presented a list of sequential compile-time invocations aimed at representing a theory. In the middle frame a (voluntarily confusing) representation of the in-memory storage of the given clauses, grouped for family membership. To the right, a human-readable theory.

4.1 Indexing

Indexes are the containers for the collections that actively holds all the `SituatedIndexedClause`. They provide the methods needed to make the system interact in a meaningful way with such data structure. There are three types of indexes, although this is not explicit in the type hierarchy that describes them. The difference leading to their conceptual birth resides in the semantic range of possible values that constitutes the first argument of the clause's head.

- **List based index -** this kind of index reflects the fact that the first[9] argument of the term that led to *situate* a clause in this particular place does not carry enough information value to differentiate one clause from another. `DirectiveNode`, `ZeroArityNode`, and `VariableIndex` all reflects this kind of situation. The clauses indexed this way are stored in some sort of list typed collection, accordingly to runtime necessities.

- **Dictionary based index -** this kind of index is needed where the range of values for the first argument is expressive enough to be compared for equality. It's the case of the `NumericIndex` and the `AtomIndex`, where the first argument is extracted as it is from the head of the clause, and used as a term itself. This is then used as a key in map, associated with all the clauses sharing the same first argument

[9] If existing at all

- **Tree based index -** the only tree based index is the `CompoundIndex`. It contains a representation of its content based on a structure that mimic the whole tree again, recurring as long as the term appearing as a first argument as in its turn a first argument that is a compound. Design-wise, this is why a `CompoundIndex` is considered as leaf when it comes to hierarchy nomenclature: only the first level of this index can be thought as a system-known node. The rest of the recursive structure constitutes the means by which a `CompoundIndex` manages its own content, to the same extent an `AtomIndex` uses a dictionary

4.2 Imposing order

As already mentioned, only certains `TopLevelNode` have the ability to understand the global ordering. Let's dive more into this concept, remembering that in fig. 13 the `TopLevelNode` are the `FunctorNode` and `ArityNode` containing an f and a 1 on the left side, and the ones containing g, 0 and 2 on the right side. We will actually focus only on the `ArityNode`, as there is where most if not all of this reasonings are executed. Let's assume a `get(`$f(a)$`:-_)` operation is inquired. All the clauses unifying over the argument clause needs to be found across the tree. The task is demanded to the f `FunctorNode`, and again to the 1 `ArityNode`. Here, the first argument of the head of searched clause is recognized as an `Atom`, which implies that both the `AtomIndex` and the `VariableIndex` have to answer their own, ordered, list of `SituatedIndexedClause`[10]. This means that at the `ArityNode` level, it is possible to order again the results[11], strip the objects from the *situatedness* and return the result to the upper layers. This is depicted in the table 2.

Partial solution	Reordering	Final output
⟨-5, $f(a)$:-_⟩	⟨-6, $f(X)$:-_⟩	$f(X)$:-_
⟨3, $f(a)$:-$g(Y)$⟩	⟨-5, $f(a)$:-_⟩	$f(a)$:-_
⟨-6, $f(X)$:-_⟩	⟨-2, $f(X)$:-*cup*⟩	$f(X)$:-*cup*
⟨-2, $f(X)$:-*cup*⟩	⟨-1, $f(Y)$:-$f(X)$⟩	$f(Y)$:-$f(X)$
⟨-1, $f(Y)$:-$f(X)$⟩	⟨3, $f(a)$:-$g(Y)$⟩	$f(a)$:-$g(Y)$

Table 2: The steps taken to solve the `get(`$f(a)$`:-_)` call.

4.3 Caching

To make execution have the ability of going even faster, every node was given a cache. When a query is detected as *global*, which means that every possible element in the sub-tree would end up as being retrieved, the content of the cache would be returned instead. This is meant to short-circuit the tree exploration as

[10] If the first argument of the query would have been a `Numeric` (resp. `Compound`, the `NumericIndex`(resp. `CompoundIndex` and the `VariableIndex` would have been inquired. In the case of a `Variable` all four branches result would have been merged

[11] The algorithm of choice is the merge sort, as the input comes as two partial ordered lists

soon as it is detected that the visit would be pointless. The cache is built along the creation phase of the tree, and invalidated as soon as a retraction is made. Invalidation is propagated to the whole interested sub-tree. It is recalculated as soon as another *global* query is detected, so that no resources are wasted in the cache management: the whole exploration would have been performed anyway.

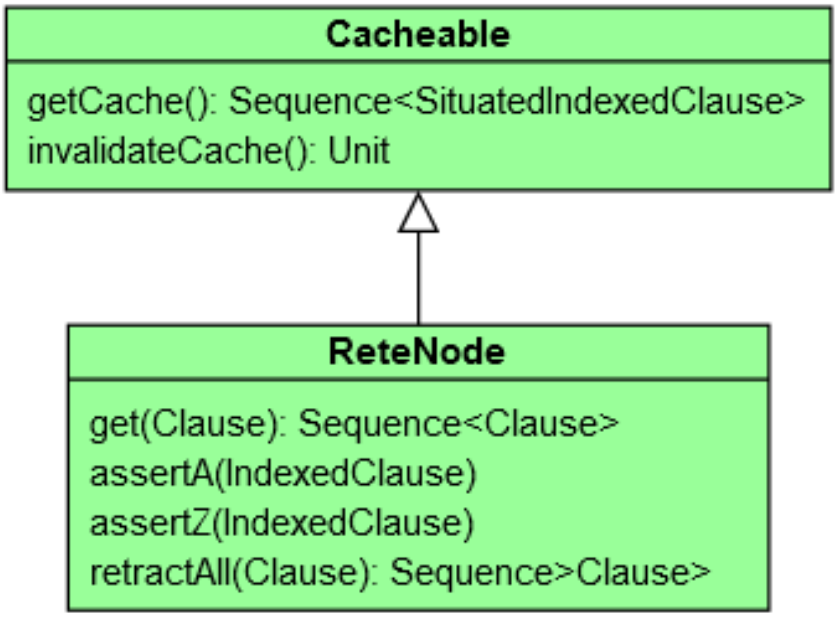

Figure 14: The `Cacheable` interface

4.4 Retraction

A retraction call removes the given `Clause` from the theory and returns a `Sequence` of the clause found. Once the ordering mechanism and the sharding of the indexes were in place, it became less inthuitive to implement, as the degrees of freedom for a clause to be situated increased along.

All For the case of the *all* case of retraction, difficulties didn't increase that much. With the same modality explained in section 4.2, the correct branches of the tree interested by the retraction invocation would be put in charge of retracting all the clauses that would unify over the given one, passing the result to the upper layers to be reordered. *All* semantic is not affected by the sharded structure the `ReteTree` has.

First Many of the functionalities the `ReteTree` possesses have been implemented for this feature alone. When the knowledge base is scattered amongst many places, the concept of *being-the-first* is pushed to another level, as it becomes a relative property. Retracting the very first `Clause` of the theory that would unify over the input from an absolute standpoint is the main reason the `TopLevel` and `Retractable` mechanisms are in place. Let's consider the left side of fig. 13, and the query `retractFirst(` f(_):-_`)`. For all the indexes there is the possibility that it might possesses the actual answer, but their word alone isn't enough, and they can't be let to choose the answer on their own. Instead, (recursively, if a deep `CompoundIndex` would be present) each of them is asked for their vision of what the first matching clause would mean for them, without actually retracting anything. As soon as the (at most) four possible result reach the related `TopLevelNode`, the rightful *first* would be chosen, and exploiting the `Retractable` capability inside the emerged winner, the `SituatedIndexedClause` is actually removed from the index, at whatever level of nesting it would be placed.

Only For all the reason explained in the paragraph above, there are no means to be sure where to retract all the clauses in bulk. Part of the solution may be in one branch, another somewhere else. This feature has then been implemented at the `RootNode` level as a `retractFirst` performed for the requested amount of times.

5 Testing

5.1 ClauseDatabase

For testing the `ClauseDatabase` it has been executed the same test suite from the previous implementation, adding the test case for the failing scenarios. The old test was maintained to be sure to not break any existing functionality, since the interface was already used in many other parts of the system.

it.unibo.tuprolog.theory	90.9% (10/ 11)	74.6% (44/ 59)	73.3% (74/ 101)
it.unibo.tuprolog.theory.impl	100% (3/ 3)	88.9% (16/ 18)	69.9% (51/ 73)

Figure 15: The coverage for the `ClauseDatabase`. Columns depicts the percentage of classes, methods and line of code tested in the package.

5.2 ClauseCollection

The collection framework has been tested for the all the occurrences any collection would encounter: absence of the element, rightful returned result, correct size after insertion, correct order, correct retraction. These are very straightforward and common tests, and there's little to state more than the actual presence and correctness.

Package ▲	Class, %	Method, %	Line, %
it.unibo.tuprolog	0% (0/ 1)	0% (0/ 1)	0% (0/ 1)
it.unibo.tuprolog.collections	84.6% (11/ 13)	53.8% (43/ 80)	58.7% (98/ 167)
it.unibo.tuprolog.collections.impl	100% (6/ 6)	100% (36/ 36)	93.8% (76/ 81)
it.unibo.tuprolog.collections.rete.custom	100% (6/ 6)	85.7% (24/ 28)	87% (47/ 54)
it.unibo.tuprolog.collections.rete.custom.clause	100% (5/ 5)	86.4% (19/ 22)	81.8% (27/ 33)
it.unibo.tuprolog.collections.rete.custom.leaf	58.8% (20/ 34)	73.8% (79/ 107)	74.3% (208/ 280)
it.unibo.tuprolog.collections.rete.custom.nodes	100% (33/ 33)	96.6% (114/ 118)	96.6% (422/ 437)

Figure 16: The coverage for the Collection framework.

5.3 ReteTree

The `ReteTree` has been tested with a generator of clauses that creates a big set of terms, that are then composed into clauses to create a whole valid theory to be fed to the data structure. In this way it has been exhaustively checked amongst many kind of possibilities without, since the datasets this generator would create are very large and variegated.

it.unibo.tuprolog.collections.rete.custom.clause	100% (5/ 5)	86.4% (19/ 22)	81.8% (27/ 33)
it.unibo.tuprolog.collections.rete.custom.leaf	58.8% (20/ 34)	73.8% (79/ 107)	73.9% (207/ 280)
it.unibo.tuprolog.collections.rete.custom.nodes	100% (33/ 33)	95.8% (113/ 118)	96.3% (421/ 437)

Figure 17: The coverage for the `ReteTree`. Columns depicts the percentage of classes, methods and line of code tested in the package.

6 Benchmark

These values are generated running *100* times each measured scenario, discarding the first one, so to produce an average value that would actually resemble one where the full system would be in its full operativity. The first result is discarded because at that point in time the JVM is still loading all its dependencies, and would react fast on method invocations. Since everything this whole sub-module does is completely delegated to the `ReteTree`, this is the only part subjected to benchmarking.

The unit of measurement may be different on different rows of the same table, so it will always be stated.

The benchmarks are performed on an Intel® Core™ i7-8750H CPU @ 2.20 GHz, with 16 GB of RAM memory installed.

Execution time is obtained with the experimental Kotlin feature `measureTime()`, which takes a whole block of code, executes it and return a `Duration` object describing the elapsed time.

Creation The `ReteTree` is created and populated at the same time with a number of clauses. There is no difference in the average execution time nor the actual implementation, so that only a table of values is showed for the *ordered* and *unordered* implementation.

# of clauses	Average time
0	11.22 μs
3890	8.92 ms
19450	41.76 ms
194500	453.70 ms

Assertion The `ReteTree` is created empty, to be later filled with some clauses. The benchmark is only performed on the *ordered* version, since the *unordered* one won't have allowed A type assertions.

# of clauses	A assertion avg.	Z assertion avg.
3890	9.28 ms	10.00 ms
19450	43.16 ms	44.61 ms
194500	425.00 ms	431.20 ms

Get all operation The following table shows the benchmark on getting *all* the clauses from the `ReteTree` without performing the actual parsing of any clause. The difference between *Cached ordered* and *Cached unordered* has been collapsed in only *Cached*, as the results were absolutely identical.

# of clauses	Cached	Not cached ordered	Not cached unordered
3890	15.07 μs	10.33 ms	3.06 ms
19450	15.73 μs	46.56 ms	11.75 ms
194500	16.71 μs	532.20 ms	89.55 ms

Get *o/1* family operation The following table shows the benchmark on getting all the clauses from the *o/1* family. This family has been chosen because it is one of the more dense in terms of compound terms, so that reordering impact and tree depth could be stress tested accordingly. The difference between *Cached ordered* and *Cached unordered* has been collapsed in only *Cached*, as the results were absolutely identical.

Total	Selected	Cached	Not cached ordered	Not cached unordered
3890	32	64.65 μs	77.48 μs	88.17 μs
19450	160	59.68 μs	112.57 μs	94.61 μs
194500	1600	66.77 μs	104.14 μs	112.68 μs

Retracting first member of *o/1* family The following table shows the benchmark on retracting the first memeber of the *o/1* family. Caching has no impact over retraction, hence it isn't included into the table

Total clauses	Retracted clauses	Ordered	Unordered
3890	1	320.01 μs	304.72 μs
19450	1	365.40 μs	365.86 μs
194500	1	363.46 μs	362.48 μs

Retracting *o/1* family operation The following table shows the benchmark on retracting all the clauses from the *o/1* family. This family has been chosen because it is one of the more dense in terms of compound terms, so that reordering impact and tree depth could be stress tested accordingly. Caching has no impact over retraction, hence it isn't included into the table

Total clauses	Retracted clauses	Ordered	Unordered
3890	32	478.27 μs	420.43 μs
19450	160	733.23 μs	676.88 μs
194500	1600	13.87 ms	10.35 ms

7 Conclusions and Future works

Abstracting away Rete from Collections The collection framework could be made generic, but that would mean engineering a way to decouple them from the concept of the Rete algorithm. While I'm sure there is a plethora of design pattern waiting to be used for this exact intent, this would in turn means reintroducing the problem of letting a generic class understanding that it is being used for a certain purpose dictated by the data type it is hosting. It is surely a complex and beneficial task, that would have gone beyond the goal of this project.

Unifier dependency When coming to a dictionary-based `IndexingLeaf`, the type of the first argument is identified, reified into its Prolog class hierarchy type, and put as it is in a map as a key, e.g. the `Numeric` 1. When looking for the associated clause, the only piece of information looked at is the key of such a map: if you look for a 1, you get a 1, as it is the only `Numeric` that would unify. This stops being true in the moment the standard unifier is overridden. This means that the unifier can be taught to unify together, for example, 1 and -1. This means that looking for a 1 should also present results from the -1 branch. While this is impossible at the moment, the key type can be changed to be a `Set` of values instead, to be answered from the overridden unifier after presenting him the value that one wants to be indexed.

Low level performance improvement A lot of smart hacks and tricks have been employed to make the `ReteTree` as fast as possible. This does not mean that they have been employed all. The following possibilities are surely open in terms of improvement.

- Short circuiting - as soon as a possibility for a short-circuit is seen, it should be taken. The `ReteTree` can expand very easily in depth and width, so that it can become extremely inefficient not capitalizing on such opportunities.
- Memory saving - the `ReteTree` is absolutely time efficient - space consuming. A more appropriate management of the dead branch can help save on memory too, and not only execution time.
- Parallelism - Having a thread pool constantly ready to split work on different branches of the tree, or performing the sorting routines mor eefficiently could vastly improve the tree efficiency.

References

[1] "tuProlog official project page." https://apice.unibo.it/xwiki/bin/view/Tuprolog/Download.

[2] "prolog2Kotlin official project page." https://pika-lab.gitlab.io/tuprolog/2p-in-kotlin/.

[3] "trustalo - a well aged oop final project." https://bitbucket.org/danysk/oop17-bonarrigo-manuel-trustalo/src/master/reports/Relazione%20OOP%20-%20Libreria%20Persistence.pdf.