Personal notes/informal code guide to the Kernel of Sequoia.

* Start with going to the Reasoner class. This is an instance of a reasoner.
* Legend: **black** are names in the program; **red** are suggestions for refactoring code, **Blue** highlights abstract procedure differences between the theoretical calculus and the implementation.

**REASONER.scala**

The reasoner stores an ontology **axiomBuffer** expressed in DLaxioms[Sequoia] (the brackets indicate that Sequoia’s own API for DL axioms are used, rather than OWL’s API), an ontology **dlOntology** in DL clausal form, a context structure **contextStructure** that is used for taxonomy calculations, a simple flag **inconsistentOntology** for consistency, and two taxonomies: one **classTaxonomy** for classes, one **individualTaxonomy** for individuals.

All these things are initialised empty.

Then the reasoner contains a bunch of methods to retrieve solutions to reasoning problems involving the ontology, and possibly some other elements of the signature, like concepts or constants. These methods develop the fields of the reasoner discussed above as necessary. For instance, classification will force the taxonomy to be computed, if it has not been so already. Loading ontology will import the ontology in DL[Sequoia] form, then populate the ontology in clausal form.

**Note:** ideally you have a single context structure that you keep modifying.

**The STRUCTURAL package**

The whole purpose of this package is to store and transform the ontology in DLaxioms[Sequoia] form into the DLclause form. The way in which this happens is a bit strange, because instead of implementing the package as a function, it is implemented as an object **DLontology** which takes an ontology **axioms** in DL[Sequoia] form as an input parameter and does the whole conversion on creation.

**DLOntology.scala**

Every instance of DLOntology itself is an object where the ontology clauses are stored in indices, with some redundancy (i.e. some clauses stored in several indices). We also keep a collection of relevant types of objects, such as syntactically unsatisfiable predicates, triggers for rules of the calculus (e.g. Succ, Pred triggers, triggers for Nom, etc).

On creation, we transform the DL[Sequoia] ontology into the clause ontology as follows:

1. First, the input ontology **axioms** is transformed into an object **PreprocessedAxiomCollection** which contains several buffers with the axioms, having encoded away role chains and transitivity, and put all class expressions in NNF, and checked that the ontology satisfies the restrictions on use of complex roles. This also **internalises the entire ABox.** This process also **silently filters unsupported axioms** by Sequoia. All items necessary for this step are stored in the subpackage **CHAINS.** Worth saying that the axioms are stored in an unusual form, as pairs of the form (complement(lhs),rhs).

1. We normalise and clausify the axioms in the buffers of **PreprocessedAxiomCollection** and for each of them immediately call the method **addSTClauseToOntology** in the result, which first transforms the ontology into a pure, normalised DL clausal form, and then calls the item **addClauseToOntology,** which scans the clause to be added to add all info to the indices, including finally adding the clause to (at least) one of the indices. In the case of nominal concepts, the normalisation replaces each nominal class {o} by a concept C\_o, and adds the clause C\_o => x= o. **By itself, this is not enough; you would also need \top \to C\_o(o). But in our calculus, we ignore adding this to the ontology, because clauses of the form \top \to A(o) are not needed in ordinary contexts. Indeed, by the completeness proof, if A(o) unifies with some ontology body together with some atom mentioning t, then there is a clause of that atom of the form S(o,t) or S(t,o) in the context for t, the non-ground form of which triggers RSucc, and hence we have S(x,y)->S(x,y) or S(y,x)->S(y,x) respectively in the root context for o, where we already have \top \to A(x) and can do any inferences here, and then backpropagate with RPred.** It might still be worth adding it for issues of redundancy, but I am not considering that at the moment.

**CHAINS** package

This package contains all methods for transforming an ontology in DL[Sequoia] form, to another DL[Sequoia] form where role chains have been eliminated, class expressions are in NNF, and unsupported axioms have been removed, ABox has been internalised, etc. Once again, the way this is done seems unusual to me. Rather than as a process, this package defines a class containing the processed axioms and initialises an instance of this class, passing the original ontology as parameter. The transformation is done on initialisation, so I cover this class first. In fact, I cover this class only, since the rest of classes can simply be considered a separate modulus.

**PreprocessedAxiomCollection.scala**

This object represents a collection of axioms in DL[Sequoia] form with no role chains, no axioms unsupported by Sequoia, all class expressions in NNF, and no violations of the OWL 2 DL profile; furthermore, GCI axioms are represented as pairs (not lhs,rhs). Axioms of different types are stored in different buffers: a buffer **internalizedAxioms** for GCIs, **aymmetricProperties, reflexiveProperties, disjointProperties, simpleSubProperties** for axioms of the corresponding form, respectively. The rest of things are not really accessed, only auxiliary for the transformation. The transformation takes place on creation, in the following steps.

1. Creates an object to track restrictions on complex roles.
2. Creates property inclusion graph **knownPropertyInclusionGraph**. Also, eliminates axioms not needed after the property inclusion graph, as well as declaration and annotation axioms. The remaining axioms are passed to a buffer **filteredAxioms.**
3. Builds the automaton (builder) **builder** for encoding away transitivity and role chains. This eliminates some additional axioms and stores some simple property inclusion axioms in the buffer **simpleSubProperties.** The rest are passed to **remainingAxioms.**
4. This step takes all axioms in **remainingAxioms** and does the relevant transformations, namely, filtering out those not supported, replacing roles according to the automaton, and writing elements in negation normal form, plus taking the complement of all elements in the left-hand side of each axiom. This fills in the five buffers discussed above. This step also internalises the ABox.

**The CONTEXT Package**

This contains the “meat” of the reasoner. This package is used to create, modify, and saturate context structures. The way in which this works, again, is a bit anti-intuitive. There is a context structure manager which creates contexts. Context are implemented as fibers, which saturate as much as possible, send messages to other contexts, and then go to sleep, until messages from other contexts wake them up. When all contexts are sleeping, the context structure is saturated. I try to cover the packages in the classes in logical order, so I start with the central class, the **ContextStructureManager.**

Each context has an input channel that receives things.

**ContextStructureManager.scala**

This represents a context structure, according to the calculus, albeit limited in many ways.

On creation, a ContextStructureManager instance uses its parameters to initialise a context structure and saturate it. The context structure can then no longer be modified, but queries can be posed to it.

What are these parameters?

* An **ontology**
* An expansion **strategy The context structure enforces that every core must be mapped to a single context, contrarily to the calculus.**
* A set of unary predicates **targetConcepts** to be classified. The saturated context structure is guaranteed to be sound and complete for every atomic query A(x)->B(x) where A and B are target concepts.
* A bunch of parameters that we can ignore in this guide, as they are debug parameters, or they **might as well have been hardcoded.**

With these parameters, the context structure introduces a context for each concept in **targetConcepts**. Each context is created as a **Proc,** one of the mysterious CSO classes. Most likely, the procedure can be thought of as a list of instructions together with some auxiliary fields; among which the **ContextState** is particularly important since it contains all relevant clauses in the context. The creation of each context uses the manager’s auxiliary method **buildContext**s. The parameters to **buildContext** that are passed in this case (other than remnant, were-hardcoded, or debug methods) are the core, a flag stating that the concept is **a root concept (i.e. a concept we’ll read queries from)**, a fresh index of context clauses specially formatted for root contexts to maintain information about the superclasses of the corresponding core concept that are derived within the context, a fresh literal order, a fresh incoming channel for this node, and a flag about whether the hornPhase is active.

The **buildContext** method is an auxiliary method invoked to create contexts. Its parameters are:

* queryConcepts: ignore this. This is here as a placeholder for future stuff.
* Core: context core
* rootContext: flag that tells whether this is a root context.
* workedOffClauseIndex: an index to store clauses already derived in the context.
* Edge: the incoming channel for this context.
* Ordering: the context order for this context.
* hornPhaseActive: a flag telling whether Horn Phase is now active or not.

The main task of the context builder is to create a **ContextState**, and then pass it on to the “makeContext” method of the Context object, which returns a procedure. It is straightforward; the only difference is that a **NominalContextState** is used when creating nominal contexts, and a **ContextState** is used otherwise.

Each context is created as a procedure. After defining it, the ContextStructureManager launches it (via *fork*). It also increases by 1 the number of active context structures, stored in **activeCount.**

To wait for saturation, ContextStructureManager starts a latch at 1, and waits for it to reach 0, then it returns. The ContextStructureManager ensures that when **activeCount** reaches 0, the latch is decreased from 1 to 0. If the Horn Phase optimisation is active, it uses two latches. When the first latch reaches 0, it sends a message to all contexts to start the nonHorn Phase, changes the HornPhase flag to false, and updates **activeCount** to be equal to the number of contexts created at the end of the Horn Phase.

One may wonder: if the contexts are processes, how and when do we retrieve the queries from them? **A side-effect based system is used.** The answer is that the ContextStructureManager stores the classification of the concepts given in the parameter **targetConcepts** in a map called **provedAtomicQueries.** For each concept in **targetConcepts,** during the creation of the corresponding context, the value (a set) of this map for the corresponding key(concept) is passed as a parameter to the Context Index for the creation of the context. That way, as this context index updates during saturation, the map **provedAtomicQueries** updates simultaneously.

Incidentally, there is a flag **contradictionDerived** which can update to “true” when an inconsistency is derived.

ContextStructureManager also contains some auxiliary methods typically called by contexts in the structure during execution which typically create or retrieve particular contexts. In particular, **getAllContexts** retrieves all contexts, **getSuccessor** retrieves a context with a particular core, **getNominalContext** retrieves the nominal context for a particular constant, and **getContextsWithNominal** retrieves all contexts that mention (so far) a particular Nominal.

Finally, ContextStructureManager has two methods to output the relevant classifications, which are extracted from **provedAtomicQueries** after saturation. **getClassesTaxonomy,** and **getIndividualsTaxonomy** retrieve the classification for the original ontology – excluding auxiliary concepts – and the hierarchy of individuals.

Natural next step here seems to be: what is a ContextState, and what sort of procedure does **makeContext** create. Even though **buildContext** creates first a **ContextState** and then passes it to **makeContext,** I can already tell that the ContextState class does nothing on execution, so I will cover first the Context object, which contains the **makeContext** “abstract”method.

**Context.scala**

This is a Scala **Object,** not a class. It essentially provides the **makeContext** method, which creates a context **Proc** with the relevant fields, including the ContextState. The remainder are auxiliary methods for makeContext.

**makeContext**

The input parameters for this are: the ContextState **state** (naturally), the **ontology** (needed to produce the new clauses), the context **order**, the **contextStructureManager** that generally calls this method, and the input channel **incoming** to this particular context. Because **Proc** is a class from an undocumented package, it is not clear how it works. I simply understand it as a bunch of code that is executed as a separate process.

The process has two phases: initialisation phase, and loop phase. Both are very similar: initialisation phase saturates the context on creation and then sends it to loop phase. Loop phase keeps the context waiting for new messages on the input channel; when it receives something, then it starts another saturation round.

We start covering the initialisation phase.

*Step 1:* apply the Core rule. If the core contains **owl:Nothing** and the context is a nominal context, we immediately declare an inconsistency and send a message to the context manager to stop the saturation for the whole context structure.

Note that whenever we want to add a conclusion to the context, we need to add it to the state, so we task the state with adding this clause to the context. Furthermore, as explained below, when the state receives the order to add a clause via method **addClauseToContext,** it stores the clause in the **todo** buffer of clauses that still need to be processed (by the inference rules).

*Step 2*: We apply Hyper for all clauses that have an empty body first. Any conclusions are added in the same way as in the previous step.

*Step 3:* We finish saturating the rest of the context using the **saturateAndPush** procedure, which I discuss next since it is of particular interest.

*Step 4*: Send a message to the Context Manager that the context is sleeping, so it decreases the number of active contexts by 1. **Isn’t this troublesome? As unlikely as it might be, at some point all contexts could be sleeping even though all of them have clauses waiting in the input channel, because all of them do Step 4 simultaneously.**

*Step 5*: Keep listening for messages, which will trigger a new round. Here are the possible messages that can be caught:

* StartNonHornPhase: this message is received by the manager when the horn Optimisation is used, and the second phase starts. It changes the flag in the state. Similarly to initialisation (core has already been done), we resolve first with non-Horn ontology clauses that have empty body, and then call **saturateAndPush** to complete the round.
* SuccPush: this message is propagated by a pusher of Succ trigger predicates in a predecessor context. First, we check if any clause has become unblocked because of this, and if so, unblocked clauses are processed, as if they had been in **resultsBuffer.** Then we compute the clause that should be added to the context because of this. Then we check if we can add it, and if so, we do so, as if it had been in **resultsBuffer**, which results in it being added to **todo**. After that, we propagate back all predClauses already in **workedOffClauses** for this predicate (all of them, if the predicate is part of the core) - note that if a propagator is has been added in this execution of the method for the predicate, no previously existing clause can be propagated, so we don’t even check. Finally, we complete the round with a call to **saturateAndPush.**
* PredPush: this message is propagated by a pusher of Pred clauses in a successor context, which propagates a bunch of predicate clauses. First we perform the corresponding substitution, because the clause has been imported exactly as it was derived in the successor. Then, we apply the Pred rule to all worked-off clauses in the current context, adding of course the result to **resultsBuffer.** We process this buffer, which adds the relevant clauses to **todo,** and complete the round with **saturateAndPush.**
* QueryPush: this message is propagated by a pusher of Query clauses in a nominal context, which propagates a bunch of query clauses. First we perform the corresponding substitution, because the clause has been imported exactly as it was derived in the successor. Then, we apply the Query/Close rule to all worked-off clauses in the current context, adding of course the result to **resultsBuffer.** We process this buffer, which adds the relevant clauses to **todo,** and complete the round with **saturateAndPush**.
* PossibleGroundFactsPush: this is a push from another context (a predecessor, or just a connection to a nominal node), about a single ground propagator. First, we filter it out if it is of the form O(o)->O(o) since it is tautological. Then, like a Succ propagator, we check if anything has been unlocked, in which case we add the relevant unblocked clauses are processed as if they had been in **resultsBuffer**, with the end result of them being stored in **todo.** We apply the relevant substitution if this is a nominal context. We then process the propagator as if it had come from **resultsBuffer,** adding the clause to **todo** at the end if it has been added. Furthermore, also as in the case of a SuccPush, we propagate all the **workedOff** clauses back which contain this element in the body ~ with the optimisation that if the propagator has actually been added, there is no need to do this step, by the same argument as in SuccPush. Finally, we complete the round via **saturateAndPush.**
* CollPush:this is a push of relevant atoms when an x=o equality has been derived in a query context into a nominal context. We first add the query context as a predecessor, and then try to add to the current context the relevant propagator, as if it had come from **resultsBuffer**. As above, if the propagator has been added, it will have been added to **todo** and we complete the round with **saturateAndPush;** if that is not the case, we propagate back all relevant query clauses. **How come there is no check to see whether clauses have been unblocked here? Do we even use blocking for query clauses?**
* CGCPush: this corresponds to the push of a certain ground clause derived in a nominal context, that is, a clause of the form T -> A. First we apply the relevant substitution, and then we process it as a candidate conclusion, as if it had come from **resultsBuffer.** If it has been added to **todo,** again we start a round. No “predecessor” is updated here, because this is a unidirectional propagation.
* ConstantMentionedPush: this is a message received in a nominal context from some other context that mentions a particular constant. After receiving this message, the context simply pushes back the certain ground clauses that mention it to this other context.
* ConstantExchange: this is received by a nominal context O(x), when a constant o has been mentioned in another nominal context A(x). We receive the constant a. We then act as if constant a has been mentioned in context O(x), so we import the CGCs that mention a from the nominal context A(x). I think this just makes nominal contexts homogeneous. But I wonder, isn’t this automatic…? Well, not really. If I derive a clause T->B(o) in context A(x), this does not really travel anywhere. If it should, this is how.

After having done this, we send a message to the context structure manager that a round has been finished and this context is sleeping.

**saturateAndPush**

The saturation is organised as follows: all clauses in the **todo** buffer, usually representing clauses that have been added to the context but not yet processed, are taken in turn and we try to apply all inference rules on it. Clauses to be added in this context are added back via **addClauseToContext,** which (remember) adds the clause to **todo** unless it is redundant or it must be blocked. Clauses that must be added to other contexts are stored in special ways, and propagated once **todo** is empty. This is done via an auxiliary method called **rulesToSaturation,** and the second bit is executed by a series of ``pushers’’ that retrieve input channels to other contexts via the **contextStructureManager,** and then send to them relevant clauses stored within **contextState** that have been derived while applying to it **rulesToSaturation**. See the description below for **rulesToSaturation,** but essentially it takes each clause in **todo,** each maximal (and second maximal, if root context) literal in it, and applies all relevant inference rules to this one. Conclusions of these rules that generate clauses within the same contexts are put back in **todo** and processed again. Conclusions that generate clauses within other context store those clauses, and wait for the pushing part of this method. Which we describe now.

* If this is a nominal context, we Push the CertainGroundClauses derived in this round.
* If some inconsistency has been detected, we push it directly to the manager, rather than some other context.
* We push the Pred clauses (and root-Pred) clauses derived in the previous **rulesToSaturation** round. **This is the second part of the Pred rule.**
* If this is a nominal context, we Push the QueryClauses derived in the last round.
* We push all the Succ clauses derived in the last round.
* We push all the r-Succ clauses derived in the last round.
* If the state is a root context, we push all root collapses derived in last round.
* We push to the manager a request for all certain ground facts for all constants for which a clause has been introduced in the last round.

These things are done with the help of the abstract methods in the object **ClausePusher.**

**rulesToSaturation**

For each clause in **todo,** this picks the clause and tries to apply all rules to this clause via **applyIntraContextRulesToSaturation.** Then it adds the clause to **workedOffClauses.** An important note here is that **todo** is separated in three buffers: HornAndEmptyBody, Horn, and nonHorn; where the second contains all unprocessed Horn clauses that are Horn but don’t have an empty body. Furthermore, if we are in the Horn phase, we only consider those clauses in **todo** that are Horn; the rest remain in **todo.**

**applyIntraContextRulesToSaturation**

Ok, here is where the interesting bit happens.

Given a clause, with the optimisation, we consider the set of maximal literals, if this is not a root context, or the set of maximal and second maximal literals, if this is a root context; this is because we are using the ordering optimisation. For each of the elements in this set, we consider all possible rules to be applied.

For each literal, we check if it fits to each of several (non-exculsive) patterns. For each pattern that fits, we perform several tests on each literal, applying the relevant rules if it corresponds, which leave the (potentially multiple) conclusions inside the **ContextState** buffer **resultsBuffer**, and then uses little method **processResultsBuffer,** which **should probably be inside ContextState, so that we only give as input the type of rule,** and removes one by one the clauses from the results buffer, and calls **processCandidateInference** in **ContextState** on them, and displays a message (on debug mode). We give a reference of the tests next, and note that all methods which apply rules and fill the **resultsBuffer** in each case are abstract methods from the object **RuleSaturator**:

* If the literal is a predicate with x, we consider all possibly inferences by the rule Hyper. To do this, we call **doAllHyperInferencesForPredicateInClause.**
* If the literal is a predicate with a function symbol, we do the third part of **Pred** rule (namely, taking a clause that has been propagated from a successor context, and applying resolution). **Sequoia splits the Pred Rule into three: 1a) flagging of a clause with a head of Predtriggers and 1b) propagating this clause into the relevant predecessor contexts 2) processing the propagated clause via doing the relevant subsitutions and adding it to the set SETNAME of propagated clauses via Pred, and 3) context clauses in the current context with function symbols search for clauses in SETNAME for which a resolution step can be applied.** The function to do this is **doThirdStepPredInferencesForPredicateInClause.**
* Similarly, if the literal is a predicate that is a trigger for RSucc, we do the third part of the **r-Pred** rule. **The r-Pred rule is split into 3 like the Pred rule.** The function to do this is **doThirdStepRootPredInferencesForPredicateInClause.**
* If the literal is a predicate that contains a function symbol or constant, we apply the equality rule, with the current predicate being the *target* of the equality. **Contrarily to the calculus, we do not apply the equality rule to x; this is only necessary in Root contexts, and to deal with that, we use a different approach.** The function to do this is **doEqInferencesForPredicateInClause.**
* If the literal is a predicate of the form B(o) or S(o,x), it can trigger the Nom rule, so we try to apply it. **Rule is not applied in nominal contexts, because it should always be blocked in there.** The function to do this is **doNomInferencesForPredicateInClause.**
* If the literal is an equality, we consider all applications of the Eq rule that use this equality as target. The function to do this is **doEqInferencesForEqualityInClause.**
* If the literal is an inequality, we consider all applications of the Eq rule that use this equality as target. The function to do this is **doEqInferencesForInequalityInClause.**

**ContextState.scala**

This class combines all auxiliary data structures for the Context process (that is, for the saturation process). On initialisation, this class does nothing.We thus cover the most important fields and methods provided by this.

Important to understand what this does is the notion of Round. This is not formally implemented in the program, but an abstraction. A round is executed by however the context has been initialised plus the **rulesToSaturation** method called by the context. The initialisation results in clauses appear in the **todo** buffer, and **rulesToSaturation** processes them, in the meantime adding more and updating the fields of **ContextState**. In the end, **todo** is empty, and clauses have been stored both in the **redundancyIndex** as well as the **workedOffClauses** set.

* Stored Sets of Clauses: we have several buffers of clauses, listed next:
  + - **todo:** clauses derived in this context for the current round.
    - **predClausesOntRound:** pred clauses (clauses with pred heads) that have been added to this context in the last (current, when this is used) round. At the end of the current round, they are propagated.
    - **queryClausesOnLastRound:** query clauses (clauses with query heads) that have been added to this context in the last (current, when this is used) round. At the end of the current round, they are propagated.
    - **resultsBuffer** auxiliary buffer that holds candidate conclusions derived by the inference rules.
    - **stored4NonHorn** clauses that are non Horn but have already been processed in the Horn round.
* Stored Set of Mentioned Constants: tools for keeping track of the constants so far mentioned in the context.
  + **introducedConstantsOnLastRound:** stores all constants which have appeared in a clause added in the last round.

**I think that whenever a clause with a constant is removed due to redundancy, we are currently not updating this buffer, e.g. if a clause top -> A(x) v B(o) is in the buffer, and we derive top -> A(x), then it should be as if o was never mentioned in this context. This should be done.**

* + **mentionedConstants:** stores all constants mentioned in this context.
* Trigger Sets:
  + **succTriggers:** predicates in maximal positions in context clauses that lead to triggering the Succ rule; for each predicate, we have how many times it appears in such maximal positions.
  + **rootSuccTriggers:** predicates in maximal positions in context clauses that lead to triggering the RSucc rule; for each predicate, we have how many times it appears in such maximal positions.
  + **rootEqualities:** predicates in maximal positions in context clauses that lead to triggering the Coll rule; for each predicate, we have how many times it appears in such maximal positions.
  + **nomRuleRoleTriggers:** predicates in maximal positions in context clauses that lead to triggering the Nom rule.
* Inconsistency flag, called **inconsistencyGuaranteed.**
* Successor Operations
  + **successors** field**:** we store a map from function symbol to the input channel to the corresponding successor. **What if two diff successors, same f symbol?**
  + **nominalCollapseSuccessors** field**:** for each nominal, we get the edge channel of the corresponding successor. **Isn’t this pointless, since we know exactly where it is?**
  + **getSuccessorOrElseUpdate** given a function symbol or constant, gets the relevant context. **What if there are two function symbols, pointing to the same context?**
* Clause Blocking
  + **blockedClauseIndex. Optimisation: clauses that have been derived when doing a resolution of two clauses with different bodies, each corresponding to a propagator by a different predecessor, become blocked and no further inferences are applied.** This context stores such clauses.
* Processing New Clauses
* **processCandidateConclusion** is the key method in here**.** This takes up a clause and the type of inference rule that produced it. Then, it filters it out if it contains tautologies in the head, and blocks it if it needs to. Next, it checks whether it is subsumed by some clause in the context already (except in the case of Core, where we already know it does not), and if not we call **addClauseToContextAndUpdateEndOfRoundPushes,** which updates the relevant redundancy indices, prepares or de-prepares any relevant push via Pred, Succ, or any Intercontext rule, and finally adds the clause to **todo. The call to preparePushesForEndOfRound should probably be here, rather than integrated within addClauseToContextAndPreparePushes,** because the program would be clearer if each method does “one thing” and one thing only.
  + **addClauseToContextAndPreparePushes** This adds a clause to the context; it has two parts: first it removes clauses that this one subsumes from both the redundancy and the final indices, and then removes unnecessary pushes for the end of the ground. **There should be a separate method for removing pushes for the end of the round, just for the sake of symmetry, and to make more clear what this class does.** I thus describe that part in particular: for each removed clause:

-It removes the clause from the **predClauses** index within the larger ContextClauseIndex.

-It removes the clase from the **queryClauses** index within the larger ContextClauseIndex.

-For each maximal SuccTrigger in the clause, it decreases the counter in **succTriggers** once for this trigger.

-For each maximal RSuccTrigger in the clause, it decreases the counter in **rootSuccTriggers** once for this trigger.

-For each maximal equality of the form x=o or o=x, if this is a root context, it decreases the counter in **rootEqualities** once for this equality.

Notice that this is symmetric to **preparePushesForEndOfRound.**

Next it adds the clause to the redundancy index.Finally, it calls **preparePushesForEndOfRound** which prepares all clauses that must be sent to other contexts at the end of the round, namely, when **todo** will finally be empty.

* + **preparePushesForEndOfRound.**  For each clause that has been added, this method does the following:

-For each constant in a maximal literal, it adds the constant to the buffer **mentionedConstants** and also updates the index of mentioned constants in the context structure manager, saying that the context with this core (the core of the **ContextState**) we are in, mentions a particular constant. Finally, it updates the buffer **introducedConstantsOnLastRound** by appending this constant.

-It removes redundant clauses from the **queryClauses** index within the larger ContextClauseIndex, and then adds this clause to the index.

-It removes redundant clauses from the **predClauses** index within the larger ContextClauseIndex, and then adds this clause to the index.

-For each maximal SuccTrigger in the clause, it increases the counter in **succTriggers** once for this trigger.

-For each maximal RSuccTrigger in the clause, it increases the counter in **rootSuccTriggers** once for this trigger.

-For each maximal equality of the form x=o or o=x, if this is a root context, it increases the counter in **rootEqualities** once for this equality.

* Predecessor Operations
  + **predecessors** field: map from pairs (inputchannel, term) to a set of predicates. The inputchannel is to the predecessor, and the term is the label in the corresponding edge. The List of predicates contains the predicates in that predecessor that mention the relevant function symbol and are in a maximal position (i.e. the corresponding K2) from the last time this was updated.
  + g**etRelevantContextStructurePredecessors**: given the body of a clause, checks which predecessors have a K2 that include this body.
  + **addContextStructurePredecessor:** given an incoming channel for the new predecessor, the term that is the corresponding label, and a predicate; update the predecessors index, considering the predicate is a new member of K2 **[Why not do this in bulk? K2 may be more than one predicate, no?]** Crucially, it not just adds the predecessor, but it also searches if all clauses in the blocked clause index which have now become unblocked. If there are such clauses, it takes them away from the index of blocked clauses, and processes them as usual, calling **addClauseToContextsAndUpdateEndOfRoundPushes.** Returns a Boolean specifying if any clause has been unblocked.

**ClausePusher.scala**

This object provides the abstract methods that are necessary for pushing messages from one context to another. We discuss each method in turn.

* *pushPredClausesDerivedInLastRound.* As the name indicates, this method takes clauses that have been derived in the last round, which have been stored in **predClausesOnLastRound.** Once taken, it adds it to **workedOffClauses,** among the pred clauses. Furthermore, if this is a nominal context, and the body contains no roles, and the head contains no variable, we don’t need to propagate it, because the clause is only relevant for query contexts, and it will already be propagated to them by RootPred and Query clauses; we make an exception for clauses with empty heads, however. Otherwise, we search the predecessors that contain the relevant elements in the body, and propagate the clause to them as a **PredPush.**

* *pushWorkedOffPredClauses.* This pushes clauses that were already in the context, when a new predecessor links to this context, so we can propagate it to them. This takes an input a predicate A, so as to only select the clauses that contain this predicate. However, if the predicate A is part of the core of the current context it is implicitly in all clauses, so we propagate them all. Otherwise, we only propagate those that contain this predicate. In nominal contexts, we use an optimisation: clauses with a head that is entirely ground will already be propagated via RPred and Query. **Shouldn’t we ensure here that only relevant clauses are propagated back?**
* *pushSuccClausesDerivedInLastRound.*  Nothing strange here, given one of the triggers for Succ marked during the round, we apply the relevant substitution to this predicate, call the **contextStructureManager** to understand who is the successor, and we push this predicate to them. **There is a long-winded explanation to the effect that we don’t choose K1 as in the calculus, but stick to the known predicates for the given term according to the ontology.**
* *pushRootSuccClausesDerivedInLastRound.*  This passes predicates selected in the last round to a nominal context. Furthermore, if this happens from a nominal o, to a nominal u, non-ground propagators must also be passed, because these are the only clauses in O(x) ~the current context predicate~ that are not regenerated from T -> O(x), non-ground propagators, or ground propagators, but the latter are automatically derived in each nominal context due to the ground propagators propagated through the context structure. In sum, each new succ predicate can be propagated to as many nominal contexts, as nominals appear on it. For each trigger, we apply the relevant substitution, query the contextManager for the relevant nominal context, and propagate it to it, as a **SuccPush**. **Furthermore, if the trigger is of the form B(o), we propagate it to all successors.**
* *pushQueryClausesDerivedInLastRound.* This propagates back to a query node that could collapse into a nominal, any clause that has a query head and such that its body consists of exactly the core of the original root node plus query propagators found in that quer context. This is the equivalent of using the Eq rule in the calculus to reduce an atom of the form A(o) in a query context to an atom of the form A(x), using a clause Gamma -> Delta vee x=o, and it complements the Coll “rule” which draws links from query contexts to atomic contexts with x=o. Of course, it only works for nominal contexts. We remove the clause from **queryClausesOnLastRound** and add it to **workedOffClauses,** on the query index. If the body of the clause is empty, we must pass it to all predecessors labelled with a core. Otherwise we pass it only to predecessors that contain some of the atoms in the body.
* *pushWorkedOffQueryClauses.* This propagates query clauses already derived to a query context. We select the relevant clauses in those from worked-off, for a given edge label, and push them all. It does this via a **QueryPush.**
* *pushRootCollapsesDerivedInLastRound.* If this detects an equality of the form x=o, it propagates the core into a nominal context. It does it via a **CollPush.**
* *pushInconsistentOntologyMessage.* Exactly what it says on the tin. **Probably should be an inline,** but not sure.
* *pushCertainGroundClausesDerivedInLastRound.* This is only triggered in nominal contexts. For all certain ground clauses derived on the last round, this propagates this to all the constant-predecessors.
* *pushWorkedOffCertainGroundCluases.* This is only triggered in nominal contexts, it complements the previous rule, by propagating those certain ground clauses already derived.
* *pushRequestAllCGCsForConstantsIntroducedInLastRound.* This sends a request to a nominal context saying: “I introduced a new constant corresponding to the one you represent; please give me all function-free certain ground clauses”. For each constant introduced in the last round, we sent the nominal context a message **ConstantMentionedPush.** Also, to ensure that every nominal context we propagate a **ConstantExchange** push to the nominal for the mentioned constant, containing this constant**.**