Enhancing RETE Algorithm to support Dynamic Typing

Authors: Peter Lin, Johan Lindberg, Joe Kutner

Date: December 2008

Draft: 4th revision

Introduction

The last few years, dynamic languages like Python, and Ruby have seen an increase in popularity. Although dynamic languages are powerful and have some advantages over statically typed languages like C++, Java and C Sharp, implementing a RETE rule engine in those languages poses some interesting challenges.

In dynamic languages, an object can differ quite a lot from it's class declaration and from other instances of the same class. This means that you can't decide, based on type, which branch of alpha nodes to pass an object to. Most programmers (of dynamic languages) use the strategy of "duck typing" to handle this sort of problem and we'll therefore discuss a Duck Type Node as a suggested replacement of the Object Type Node.

Rete algorithm

The RETE Algorithm is an efficient pattern matching algorithm that is used in many Production Rule Systems and Business Rule Engines (BREs). The RETE algorithm was designed by Charles Forgy in the 1970s and is described in his 1979 PhD thesis[1] as well as in an article published in Artificial Intelligence Journal 19 from 1982[2]. The RETE algorithm reduces the computation required to evaluate a set of rules and update the list of activations when facts are added to, or removed from the working memory.

It creates a discrimination network based on the rules of the system. When facts are added (asserted) or removed (retracted) from the working memory they pass through the RETE network. When a rule matches completely, the engine creates an activation instance for the rule and adds it to the conflict set, which is commonly called the agenda. When a fact is removed from the working memory, the rule engine must remove the activations associated with the fact.

Dr. Forgy's 1982 paper describes 3 main types of nodes: 1-input, 2-input and terminal. One input nodes are responsible for evaluating a single fact for one condition. The condition can be a literal constraint like `name == bob` or perform string matching like `name.indexOf("new york") > -1`. Two input nodes are commonly called beta node or join node. They perform joins between 2 or more facts. Terminal nodes are specialized 1-input nodes and are responsible for creating the rule activation and adding it to the agenda. This is what makes the RETE algorithm efficient compared to a naive implementation that tries to calculate the list of activations in a Production System.

The RETE network begins with a Root node where all facts enter the network. The facts are immediately passed to an Object-type node. The job of an Object-type node is to make sure the alpha nodes following it only evaluate conditions for that type. It does this by filtering the facts based on the object-type. A simple RETE implementation would attempt to propagate the facts to every child of the root node. An optimal root node would lookup the object type node by type and only propagate the fact to those nodes.

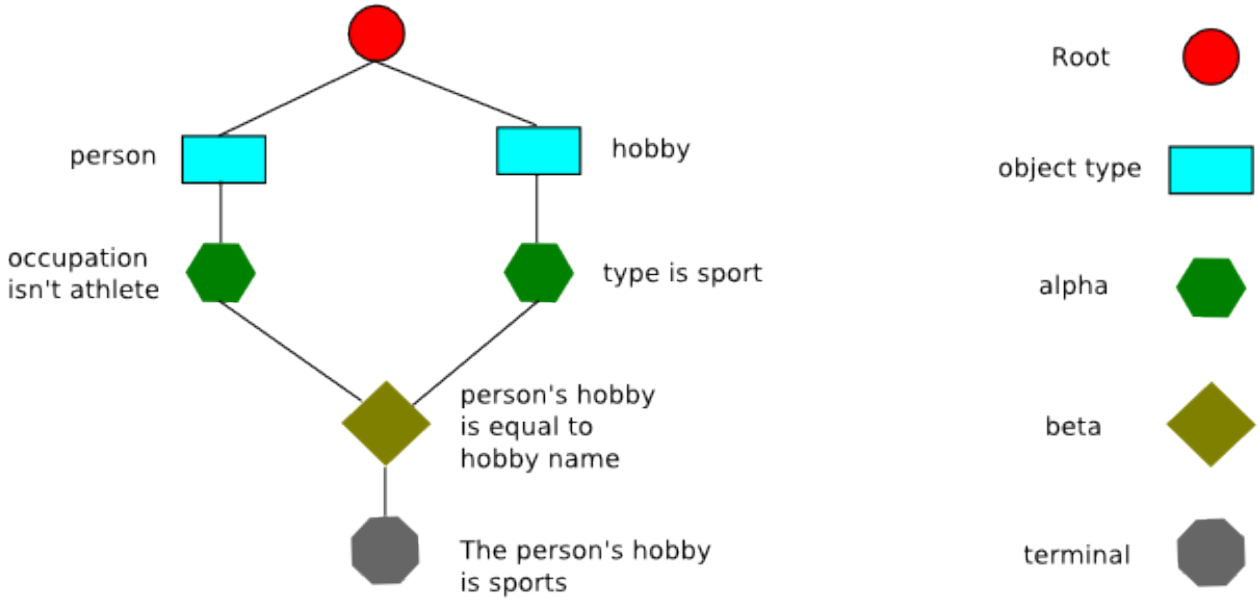


Diagram 1

Object-type nodes can have alpha, beta or terminal nodes for children. Some implementations like JESS and Jamocha utilize a left-input-adapter node to connect an alpha node to beta node. This means Object-type nodes can also have left-input-adapter nodes for children. Alpha nodes can have alpha, beta, and terminal node for children. With JESS and Jamocha, the alpha node can also have left-input-adapter nodes for children. Beta nodes can have beta or terminal nodes for children.

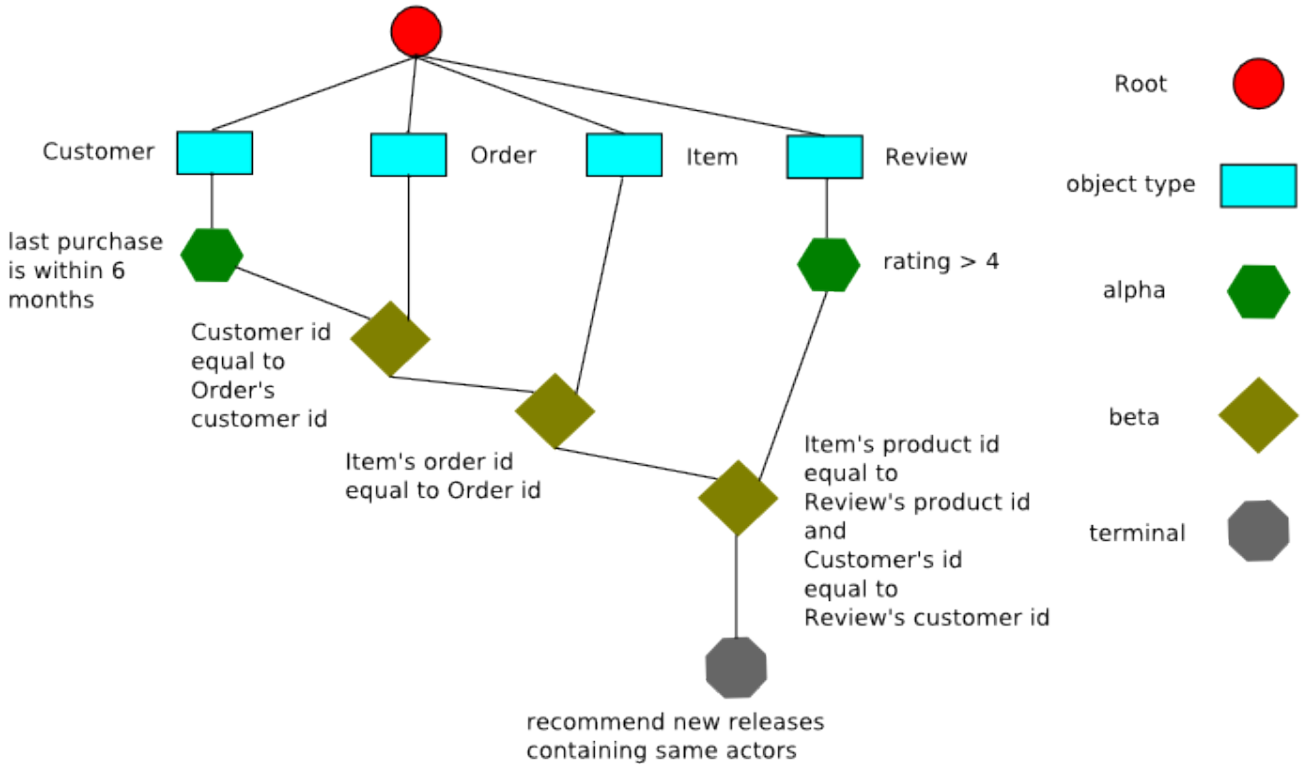


Diagram 2

Here is an example RETE network for 2 simple rule in Jamocha. Notice both rules do not join transaction with

rating. This means the join node simply propagates the facts down the network without evaluating them.

```
(defrule joinrule1 (declare (salience 100)(rule-version a1.1) )
      (transaction
            (countryCode "US")
            (accountId "acc1")
            (exchange "exchange1")
      )
      (rating
            (issuer "BOB")
      )
=>
      (printout t "joinrule2 was fired" crlf)
)
(defrule joinrule2 (declare (salience 10)(rule-version a1.0) )
      (transaction
            (countryCode "US")
            (accountId "acc1")
            (exchange "exchange1")
      )
      (rating
            (issuer "COC")
      )
=>
      (printout t "joinrule1 was fired" crlf)
)
```

The network for the rules include the left-input adapter node, which triggers the left input on the join node. Note both rules share the same sequence of alpha nodes and the left-input adapter node has 2 children.

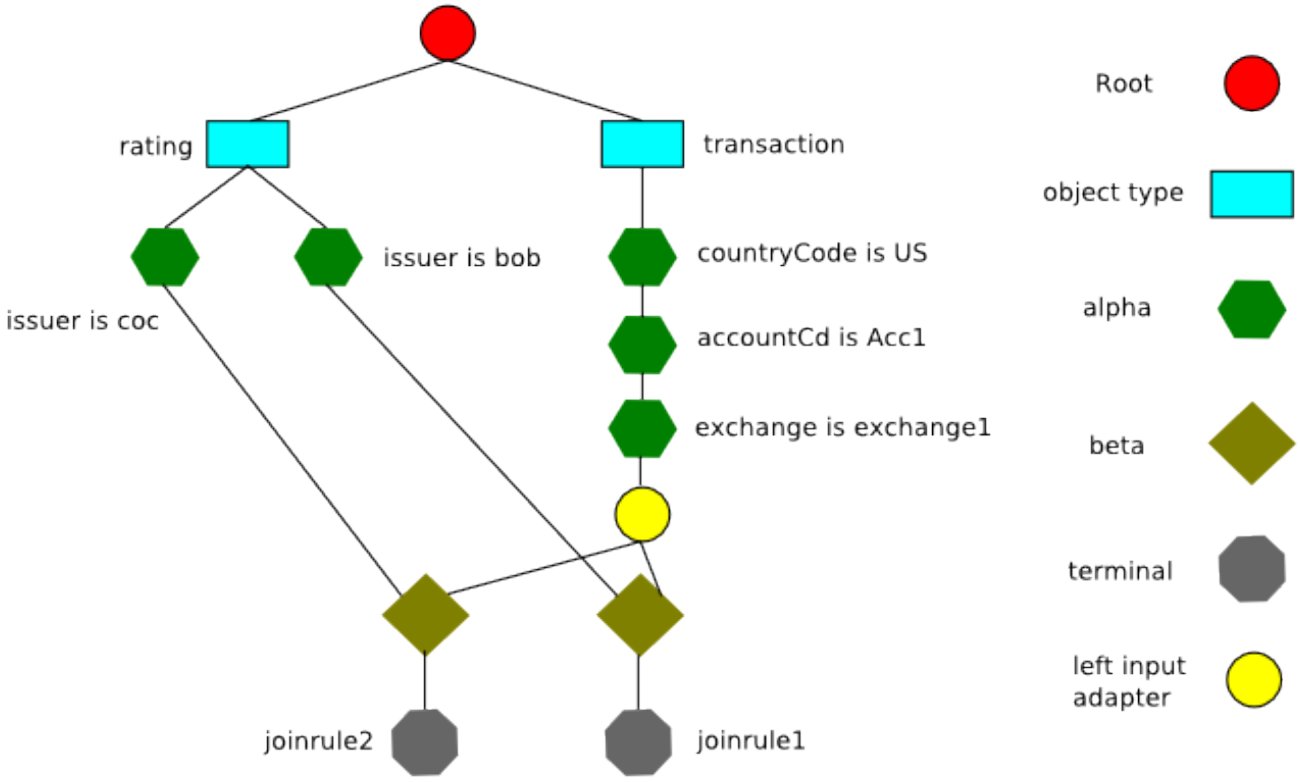


Diagram 3

Each node has a memory, which remembers which facts have passed evaluation. Each alpha node has a corresponding alpha memory. Each beta node has 1 memory for each input. Typically, the left input has a beta memory and the right input has an alpha memory. The left input takes a list of facts as the input, which means the beta memory contains a list of lists. The right input takes a single fact, so it contains a list of facts. The terminal node has a terminal memory, which is a list of lists. In Jamocha, the left-input-adapter node does not have a memory. It's sole purpose is to make it easier to compile rules and active the left input on a beta node. Once a rule matches fully, the terminal node adds it to the agenda. The rule activations in the agenda is called the conflict-set. In many expert system shells and business rule engines, the conflict resolution strategy determines which rules execute first. The most common strategies is breadth and depth.

Dynamic Typing and Duck Typing

A programming language is said to use dynamic typing if it performs its type checks at run-time rather than at compile time. Python and Ruby are dynamically typed languages, Java is statically typed.

Dynamic typing has to do with variables (names) in programs, meaning that a variable can refer to any type of value. In order to handle this situation effectively programmers of dynamically typed languages have come up with a strategy called Duck typing. The essence of Duck typing is that you shouldn't check for a particular type, instead you should check that the necessary characteristics are in place. This is usually summed up in "*If it walks like a duck and quacks like a duck, I would call it a duck.*"

A rule engine that allows Duck typing would let you to write a rule such as:

```
(defrule foo
      ?type <- (type (bar ?bar))
=>
      (printout t ?type ".bar = " ?bar crlf))
```

which would match any fact that has a slot called bar.

Object Type Nodes vs Duck Type Nodes

Object Type Node

The Object type node is importance to RETE network and avoids performing unnecessary operations. Unlike a simple procedural rule engine which evaluates every object against every rule, RETE only evaluates a condition if the object type matches. The Object type node prevents this from happening by passing the facts only to the nodes that apply to them. For example, say we have two classes: customer and address. All conditions for addresses are grouped under the address object-type node. All conditions for customer are grouped under the customer object-type node. In most implementations, this is done by creating a list of valid Object type nodes and hashing them by a given Class.

Some implementations employ additional optimizations like alpha node hashing to make the object type node more efficient. Engines that implement the optimization can provide near constant performance for large rulesets with thousands of simple rules. The optimization takes advantage of the fact that object definitions generally do not have thousands of attributes and rules generally do not reason over all the attributes. Instead, it's more common to have object definitions with several dozen attributes and rules that reason over a subset of the attributes. The object type node optimization in Jamocha creates a hash from the slot + operator + value. Of the possible operators, only 2 are hashed: equal to and NIL. Here is the pseudo code for adding successor nodes to the object type node

```
addNode(Node node)
      if node not in children list
            if node is terminal or join node
                  add node to unhashed node list
            else
                  if operator is equal to
                        create node hash for key
                        add node to hashmap
                  else
                        add node to unhashed node list
                  end
            end
            add node to children list
            propogate facts to node
      end
end
```

At runtime, the node checks to see how many children it has before propagating the fact down the network. Here is the pseudo code for the algorithm.

```
assert(Fact fact)
      if slot count > 0 and children list size > (slot count * 2)
            foreach slot in the deftemplate
                  if slot use count > 0
                        create hash using "slot + equal to + value"
                        look in hashmap for alpha node
                        if alpha node exists, then propagate
                        create hash using "slot + equal to + NIL"
                        look in hashmap for alpha node
                        if alpha node exists, then propagate
                  end
            end
            propagate fact to unhashed node list
      else
            propagate fact to all children
      end
end
```

The reason this approach works is the object-type node looks up a fixed number of nodes, rather than iterating over all children and propagating the facts. Even in situations where some of the rules perform range checks like “age < 30”, the performance degradation will be greatly improved. If a rule has range and equality conditions, the engine can sort the conditions and put the equality conditions at the top. In situations where the rules are tabular like excel spreadsheets, object-type node optimization provides a significant improvement.

Duck Type

A duck typing system is more concerned with the attributes of an object than the class inheritance or hierarchy. This has several implications for discrimination networks like RETE. The definition of a class or even an object in a duck typed language can be dynamically altered at runtime. Thus, the object type is not reliable. This presents a unique problem; the rule engine must be aware of any changes to the definition of the objects in working memory. The second is “how do we build an efficient network?”

There are two potential solutions to this problem. The first is to notify the rule engine when the object type definition is modified. When a new attribute is added to the object definition, we notify the rule engine. If an attribute is removed from the object, the rule engine should mark the attribute deleted. We do this for a few reasons. The first is the rules may be using that object attribute. Second is the engine may not be able to automatically remove those nodes from the network.

From a logical perspective, if we remove an attribute from an object, any rule that uses that attribute can no longer fire. If the rule engine were to ignore the deletion and continue to fire the rule, it wouldn't make any sense. When the rule engine is notified of the change, it can scan the rules and mark it inactive or remove all rules that use the object attribute. The other thing the rule engine must do is retract all facts of that type and re-assert it. Typically, this is done with a modify function. It's important the rule engine modifies the facts of the modified type to make sure invalid activations are removed from the agenda. If the engine doesn't modify the facts, it could produce incorrect results.

The second and more complex scenario is the object definition doesn't change, but the object instance is modified. In those situations, the first approach doesn't help, so we have to find an alternative solution. Take the following business rule as an example.

```
Rule "likes jackie chan"

If
      The customer has purchased 5 or more movies with jackie chan
      and
      The customer has reviewed 2 or more movies with a value of 3 or higher
      and
      There is a new release starring jackie chan
then
      show the customer the movie and give a 5% discount
```

There are many ways to translate the business rules into executable code. For this paper, we'll go with CLIPS syntax, since it will run in CLIPS, JESS, Haley or Jamocha.

```
(defrule get_movie_purchases
      (Customer
            (accountId ?accid)
      )
      ;; we haven't loaded MoviesPurchased fact from the database
      (not
            (MoviesPurchased
                  (accountId ?accid)
            )
      )
=>
      (bind ?movies (movie-query ?accid "jackie chan") )
      (assert ?movies)
)
(defrule get_movie_reviews
      (Customer
            (accountId ?accid)
      )
      ;; we haven't loaded the review summary
      (not
            (ProductReviewSummary
                  (accountId ?accid)
            )
      )
=>
      ;; query the database for review summary with jackie chan
```

```
      ;; and rating of 3 or higher
      (bind ?review (review-query ?accid "jackie chan" 3) )
      (assert ?review)
)
(defrule likes_jackie_chan
      (Customer
            (accountId ?accid)
      )
      (MoviesPurchased
            (accountId ?accid)
            (count ?count&:(> 5 ?count) )
      )
      (ProductReviewSummary
            (accountId ?accid)
            (lowValue ?low&:(>= 3 ?low) )
      )
      (NewRelease
            (productId ?prodId)
            (starringActors $?actors&:(member$ "jackie chan" $?actors) )
      )
=>
      ;; call function to show productId
      (show-discount ?prodId)
)
```

Here are the definitions for the classes used by the example rule above.

```
public class Customer {
      String accountId;
      String firstName;
      String middleName;
      String lastName;
      String city;
      String state;
      String zip;
      int age;
      String gender;
}
public class MoviesPurchased {
      String accountId;
      List movies;
      int count;
}
public class Movie {
      List actors;
      List directors;
      Date releaseDate;
}
public class ProductReviewSummary {
      String accountId;
      int highValue;
      int lowValue;
      int averageValue;
      int count;
}
public class NewRelease {
      String productId;
```

```
      String nameOrTitle;
      Date releaseDate;
      BigDecimal releasePrice;
      List starringActors;
      List directors;
      String distributor;
      double popularityRating;
}
```

At T_1, we load the object definitions and rules. If we assert a customer fact for “Peter Lin”, the engine will do the necessary work and fire the rules. At T_2, we delete “lowValue” from ProductReviewSummary instance. If we retract the facts and re-assert, what happens? The simple answer is “likes_jackie_chan” rule doesn't fire. Since the instance no longer has “lowValue”, the condition will never match for this specific instance. At T_3, we re-add “lowValue” back to the instance we modified at T_2. If we retract and re-assert the facts, “likes_jackie_chan” fires again. If we load the facts for a different account at T_2, the rules could fire. To make things more interesting. Say we garbage collect the old facts and reload it fresh from the database instead of re-adding “lowValue”. Once again, “likes_jackie_chan” would fire. This is because a fresh instance of ProductReviewSummary would have “lowValue” attribute.

Back to the original question, how do we handle duck typing? The second approach is to create a duck type node instead of an object type node. What would the duck type node look like and how does it work at runtime? The first question we have to ask is this, “do we ignore the object definition completely?” Depending on the answer, there's 2 potential ways of implementing a duck type node. We'll take tackle duck-type node first and then move onto object-type + duck-type.

Duck Type Nodes

If we ignore the object type completely, the duck type node would only look at the attributes. This has several implications. The first issue is how should the user write the rule? If the object type isn't important, one could argue the rule syntax shouldn't care about the object name. Instead of declaring the object type, the rule should be more generic. Here's an example of the rule.

```
(defrule middle_age_male_customer
      (object
            (gender “m”)
            (age ?age&:(> ?age 50) )
      )
=>
      (printout t “the object is male and over 50” crlf)
)
```

Assuming we use “object” instead of the type name, the next question we have to ask is “what should the network topology look like?” Say we have the following rules.

```
(defrule middle_age_male_customer
      (Object
            (gender “m”)
            (age ?age&:(> ?age 50) )
```

```
      )
=>
      (printout t "the object is male and over 50" crlf)
)
(defrule middle_age_football_fan
      (Object
            (gender "m")
            (age ?age&:(> ?age 50) )
            (hobby "football")
      )
=>
      (printout t "the person is over 50 and likes football" crlf)
)
```

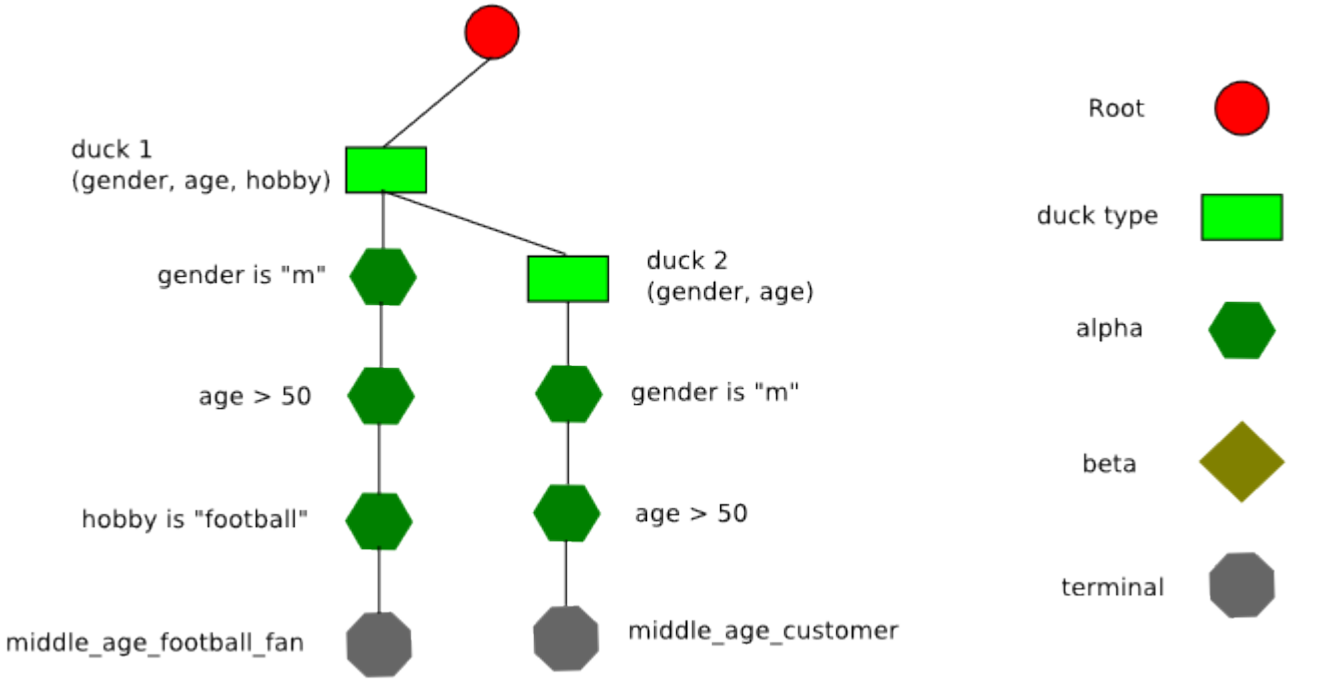


Diagram 4

Notice duck type node "duck2" is a child of duck type node "duck1". The idea is pretty simple. When the rule engine compiles the rules, it should analyze the patterns to define the duck type nodes. To do this, the rule needs to compare the attributes used by each rule. Any object pattern that is a superset of another object pattern should have the subset as a child node. Organizing the network topology this way should reduce the number of nodes the engine needs to evaluate at runtime. Unlike a standard object type node, a duck type node can have alpha, beta, terminal and duck type node for children. The equivalent rule in CLIPS format would look like this.

```
(deftemplate person
      (slot gender)
      (slot age)
```

```
      (slot hobby)
      (slot first)
      (slot last)
)
(defrule middle_age_customer
      (person
            (gender "m")
            (age ?age&:(> ?age 50) )
      )
=>
      (printout t "the person is male and over 50" crlf)
)
(defrule middle_age_football_fan
      (person
            (gender "m")
            (age ?age&:(> ?age 50) )
            (hobby "football")
      )
=>
      (printout t "the person is male and over 50" crlf)
)
```

Sharing nodes and duck typing

One common optimization with RETE is sharing alpha and beta nodes. Most commercial rule engines implement alpha node sharing. Few implement beta node sharing. Usually, when many rules share conditional elements, it means the rule needs to be rewritten to use chaining. Often this happens with rules that are very long and written improperly. Sharing nodes with duck type nodes is a little bit more complex than standard RETE. For example, we can optimize the example in diagram 4.

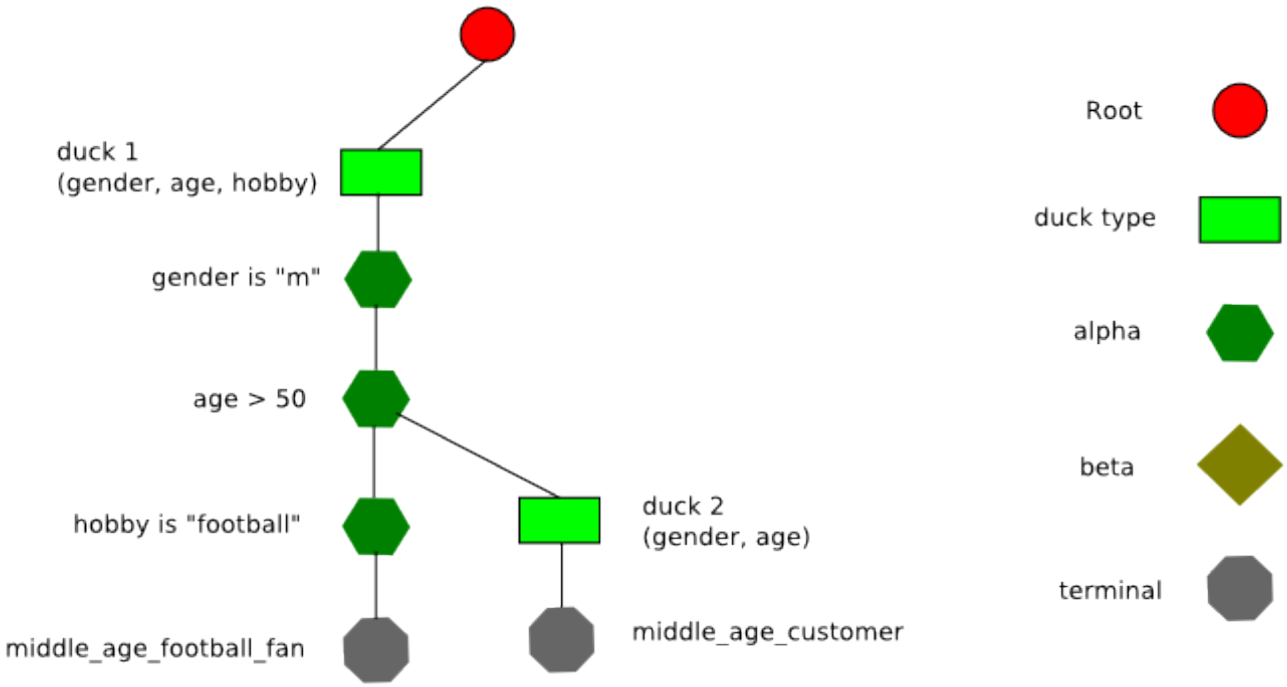


Diagram 5: Duck sharing 1

What we did was remove the duplicate nodes and attach duck2 node to “age > 50”. We can optimize it a bit more and get rid of duck2 node.

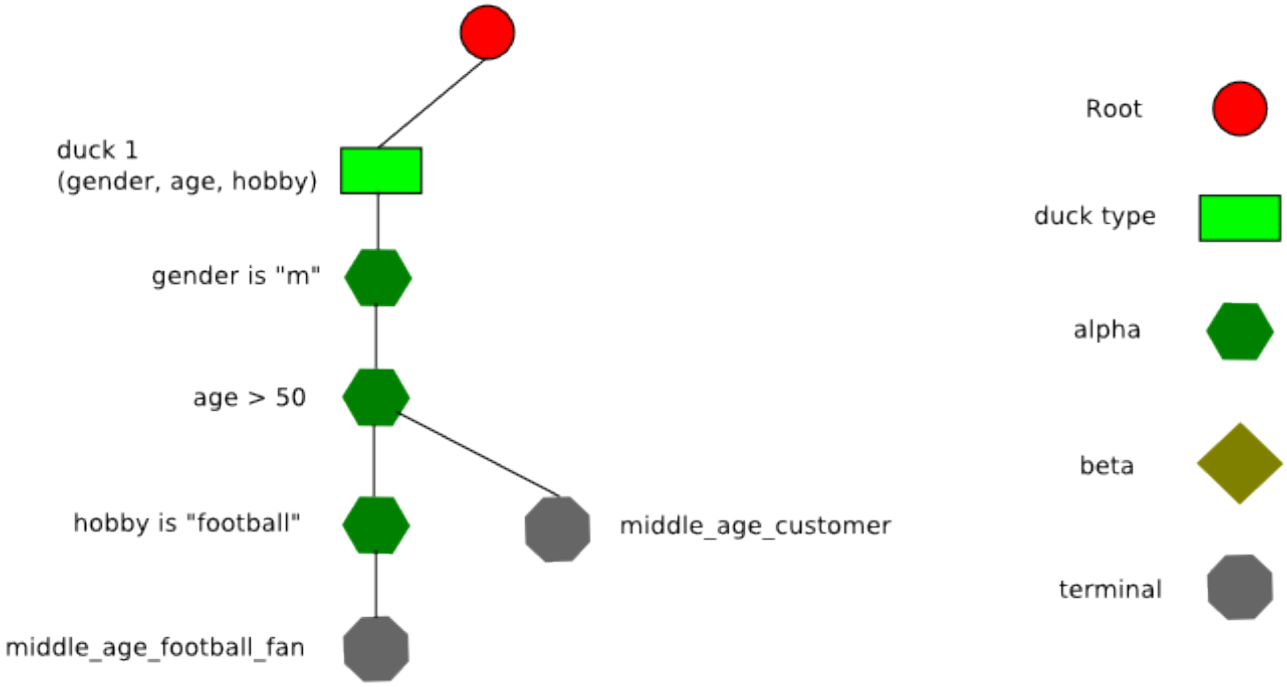


Diagram 6: Duck sharing 2

Sharing nodes in duck-type RETE is more challenging than statically typed, but it can provide significant performance improvement. Depending on rules, additional levels of optimization may not be desirable. If there are a lot of rules for duck2 node, we wouldn't be able to remove that node from the network.

Object Type + Duck Type

If we combine the object type with the duck type, we can do some odd ball things. Any fact of the same object type and fact with the same attributes will propagate down its children. Using duck-type + object-type node, “middle_age_male_customer” rule could produce “unexpected” results. For example, say we have two objects Customer and User.

```
public class Customer {
	String accountId;
	String firstName;
	String middleName;
	String lastName;
	String city;
	String state;
	String zip;
	int age;
	String gender;
}
public class User {
	String accountId;
	String firstName;
	String middleName;
	String lastName;
	String city;
	String state;
	String zip;
	int age;
	String gender;
	String username;
	String division;
	String subdivision;
}
```

If we assert an User that's male and over 50, “middle_age_male_customer” rule would fire since User has all the same attributes as Customer type. The network wouldn't look different to standard RETE. The real difference is what the object-type/duck-type node does at runtime. Here is the pseudocode for the duck type node.

```
If fact type == object type
	foreach child in children
		child assert fact
	end
else
	foreach fact type attribute
		if fact type attribute != object type attribute then
			break

		foreach child in children
			child assert fact
		end
	end
end
```

In a standard RETE implementation, the object type node would only propagate the fact if the fact had the same object type. Basically, the code in red is the extra bit of logic. One important thing to note is the root node has to propagate all facts to all object-type nodes. Each object-type then checks if a given fact should propagate to its children. For a large ruleset, the impact could be significant.

Dynamic typing

On a higher level, dynamic languages allow you to define an attribute as object type. This makes evaluating conditions and joins more challenging. If we try to join a string against a long value, should it pass? If we declare a condition that says “age > 20” and we assert age as a string, should the condition evaluate to true? To answer some of these questions, we can look at JESS. Here is an example that works in JESS and Jamocha.

```
(deftemplate person
      (slot first)
      (slot middle)
      (slot last)
      (slot age)
      (slot gender)
)
(defrule over_30
      (person
            (first ?first)
            (last ?last)
            (age ?age&:(> ?age 30) )
      )
=>
      (printout t "over_30 " ?first " " ?last " is over 20" crlf)
)
(assert (person (first "michael")(last "jordan")(age "42") ) )
```

Notice the text highlighted in red. The rule declares a pattern “age greater than 30”. The asserted fact has string “42”. If we execute “run” command, here's what JESS does.

```
Jess, the Java Expert System Shell
Copyright (C) 1998 E.J. Friedman Hill and the Sandia Corporation
Jess Version 6.0 12/7/2001
Jess> (deftemplate person
(slot first)
(slot middle)
(slot last)
(slot age)
(slot gender)
)
(defrule over_30
(person
(first ?first)
(last ?last)
(age ?age&:(> ?age 30) )
)
=>
(printout t "over_30 " ?first " " ?last " is over 20" crlf)
)
(assert (person (first "michael")(last "jordan")(age "42") ) )
TRUE
Jess> TRUE
Jess> <Fact-0>
Jess> (run)
over_30 michael jordan is over 20
1
Jess>
```

JESS coerces the string to a numeric type and evaluates the condition. This is done automatically for numeric comparisons like “>, <, >=, <=”. This type of situation can also occur in python, ruby and other dynamic languages. In JESS, type coercion only works for some situations. If we change the example slightly, JESS won't fire the rule.

```
Jess, the Java Expert System Shell
Copyright (C) 1998 E.J. Friedman Hill and the Sandia Corporation
Jess Version 6.0 12/7/2001
Jess> (deftemplate person
(slot first)
(slot middle)
(slot last)
(slot age)
(slot gender)
)
(defrule age_is_30
(person
(first ?first)
(last ?last)
(age "30")
)
=>
(printout t "age_is_30 " ?first " " ?last " is over 20" crlf)
)
(assert (person (first "michael")(last "jordan")(age 30) ) )
TRUE
Jess> TRUE
Jess> <Fact-0>
Jess> (run)
0
Jess>
```

If we run “age_is_30” in Jamocha, it runs just fine. One thing to note, not all expert system shells support type coercion. In CLIPS 6.x, the engine gives us a warning.

```
CLIPS> (assert (person (first "mike")(last "jordan")(age "42") ) )
[ARGACCES5] Function > expected argument #1 to be of type integer or float
[FACTMCH1] This error occurred in the fact pattern network
 Currently active fact: (person (first "mike") (middle nil) (last "jordan") (age
"42") (gender nil))
 Problem resides in slot age
 Of pattern #1 in rule(s):
 over_30
```

In business rule engines like Blaze and JRules, the system wouldn't allow it. For example, if we define the following Java object and try to write a rule with numeric comparison for age, it might give us an error.

```
public class Person {
private String firstName;
private String middleName;
private String lastName;
private Object age;
private String gender;
}
```

The behavior of Blaze and JRules follows Java language specification and enforces static tying. Even if we by-pass the editor and write the rule in notepad, it wouldn't be valid. In contrast, languages like python and ruby don't have these specific Java limitations, so a rule engine implemented with a dynamic language has to handle these situations. The type coercion provided by JESS provides a good model for dynamic languages. Although one could extend type coercion to handle a wider range of situations, JESS provides a great foundation. It should be noted that type coercion is implemented differently between Python and Ruby. In Ruby, a lot of type coercion is provided out of the box, whereas Python does not. This means the rule engine would need to explicitly coerce the value before evaluating it.

Potential Uses of Duck Typing

One area where duck typing could be useful is in knowledge base applications, machine learning and adaptive systems. Here's an example to illustrate.

```
(deftemplate person
      (slot first)
      (slot middle)
      (slot last)
      (slot age)
      (slot gender)
      (slot dateOfBirth)
)
(deftemplate doctor
      (slot first)
      (slot middle)
      (slot last)
      (slot age)
      (slot gender)
      (slot dateOfBirth)
      (slot degree)
      (slot specialty)
)
(deftemplate review
      (slot reviewerFirst)
      (slot reviewerMiddle)
      (slot reviewerLast)
      (slot username)
      (slot dateOfBirth)
      (slot subject)
      (slot text)
      (slot datePosted)
      (slot articleTitle)
      (slot articleAuthor)
)
(defrule reviews_found
      (object
            (first ?first)
            (last ?last)
            (dateOfBirth ?dob)
      )
      (review
            (reviewerFirst ?first)
            (reviewerLast ?last)
            (dateOfBirth ?dob)
            (articleTitle ?title)
            (articleAuthor ?paperauthor)
```

```
      )
=>
      (printout t ?first " " ?last " reviewed " ?title " by " ?paperauthor crlf)
)
```

Say we're searching the Internet for articles and reviews. John doe reviewed the paper, but it's posted on two different websites. One website is a medical forum, which contains the doctor's profile. A second website republished the review of the paper, but we only know the name and date of birth. If the rule engine supports duck typing, the rule would fire twice for the same article.

```
(person (first "john")(middle "")(last "doe")(age 40)(gender "m")(dateOfBirth
"4/5/1968") )
(doctor (first "john")(middle "")(last "doe")(age 40)(gender "m")(dateOfBirth
"4/5/1968")(degree "Md")(specialty "heart disease") )
(review (reviewerFirst "john")(reviewerLast "doe")(dateOfBirth
"4/5/1968")(articleTitle "reducing stress")(articleAuthor "Doctor Strange") )
```

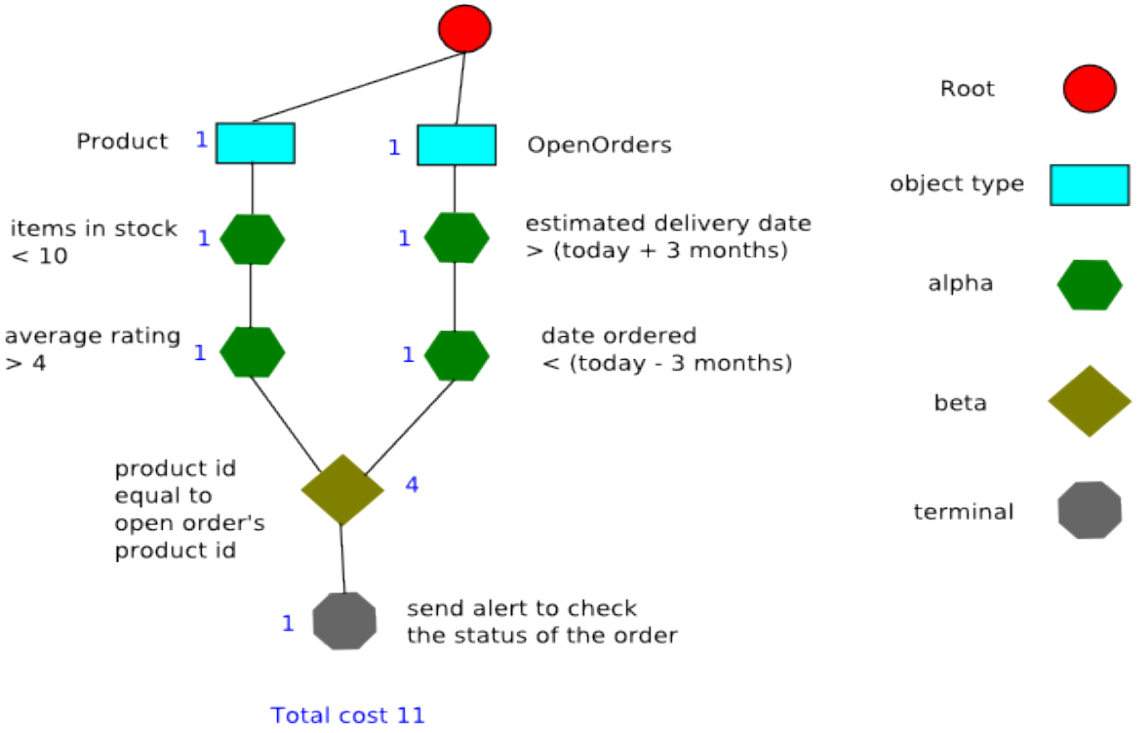
The benefit of duck typing is it reduces the number of rules I need to write. To do the same thing with static typing, I would need to know how many object types I need and what the hierarchy should be. In a semantic web, or machine learning situation, we don't know the object model, so it's impossible to enumerate all the rules. Another type of use case which may benefit from duck-type RETE is data mining or knowledge discovery in databases. In the data mining situation, we don't know the patterns at the beginning. Instead, we have meta-rules which create new rules at runtime. For this type of dynamic application, duck-type RETE can make it easier for the system.

Topology Cost Function

One way to measure the potential impact of different network topologies between standard RETE and duck-type RETE is to calculate the cost for a given rule. Once we have that, we can compare it to the equivalent rule in duck type RETE. There are several ways of calculating the cost and some papers from the 80's and 90's have tried to address these issues. The most notable cost function from the 90's is "Optimized rule condition testing in Ariel using Gator networks" paper by Eric Hanson and Mohammed Hasan. The approach Hanson and Hasan used was based on memory and IO, which were largely due to hardware and software limitations.

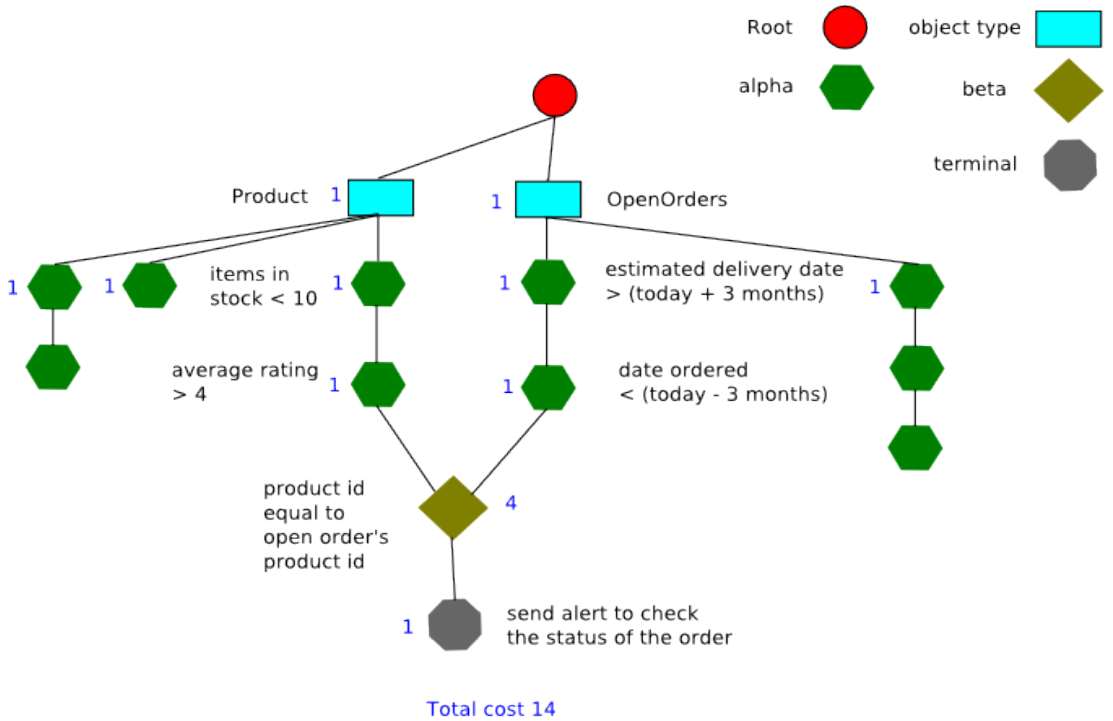
There are several reasons why Gator's cost function isn't applicable. The first is that we're primarily concerned with network topology. The second is their cost function only apply to naive RETE implementations that iterate over the facts to evaluate conditions and joins. Most modern RETE implementations utilize hashed memory, so Gator's function isn't applicable.

There's a couple of reasons why the cost function focuses on topology. Depending on the language and skill of the programmer, the efficiency of the implementation varies significantly. For example, an engine implemented in C like CLIPS can dramatically reduce the memory foot print, whereas an engine implemented in Java does not have access to pointers. Even engines implemented in Java show a wide range of performance characteristics. For the cost function to be useful, it should be application to engines regardless of the language or skill of the programmer. The simplest way to demonstrate the cost function is to mark up some examples.



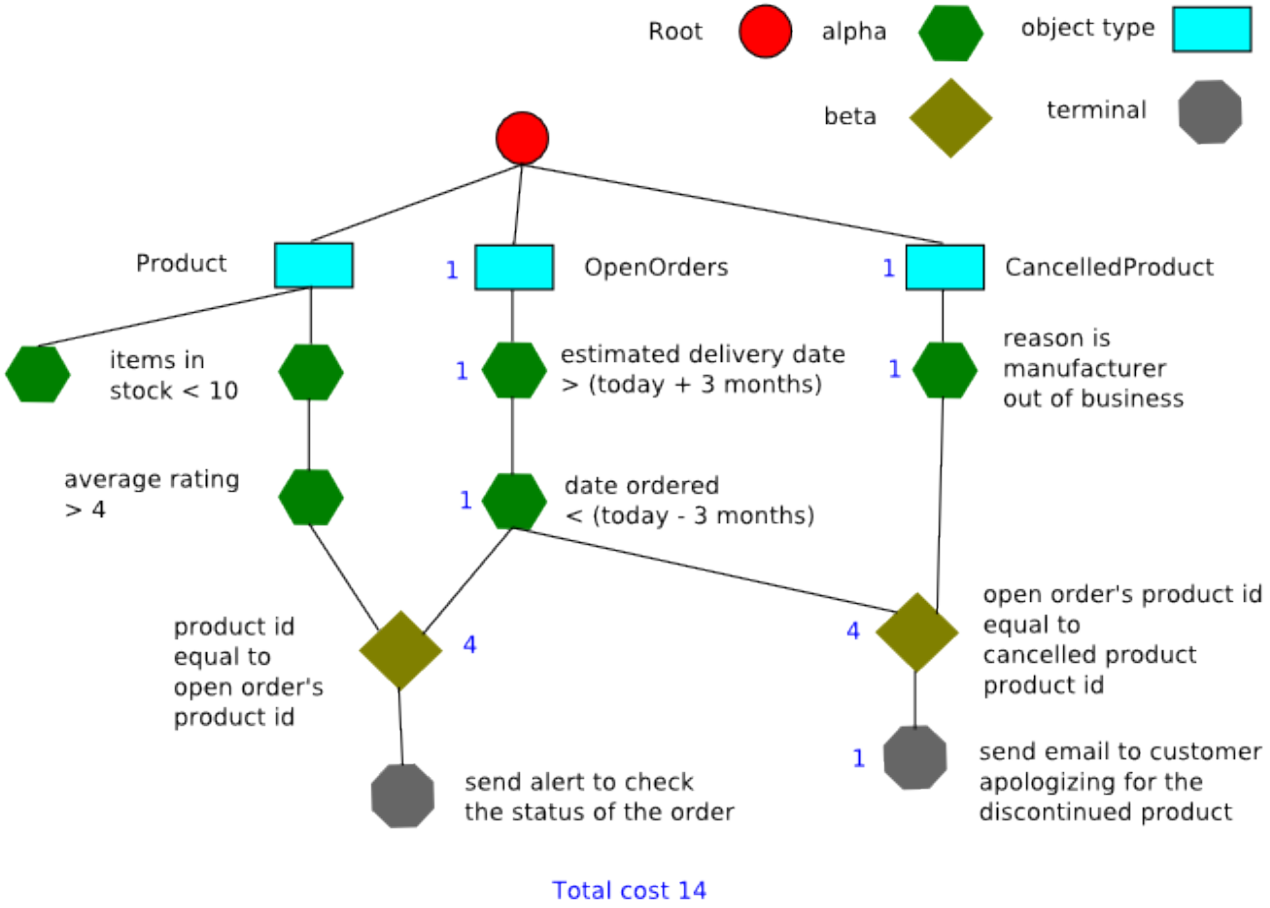
Cost Diagram 1: A single rule

The cost for the first example is 11. In this situation, there's only 1 rule, so the cost is just the nodes for the rule. The root node is excluded from the cost function since it's not a distinguishing factor between standard RETE and duck type RETE.



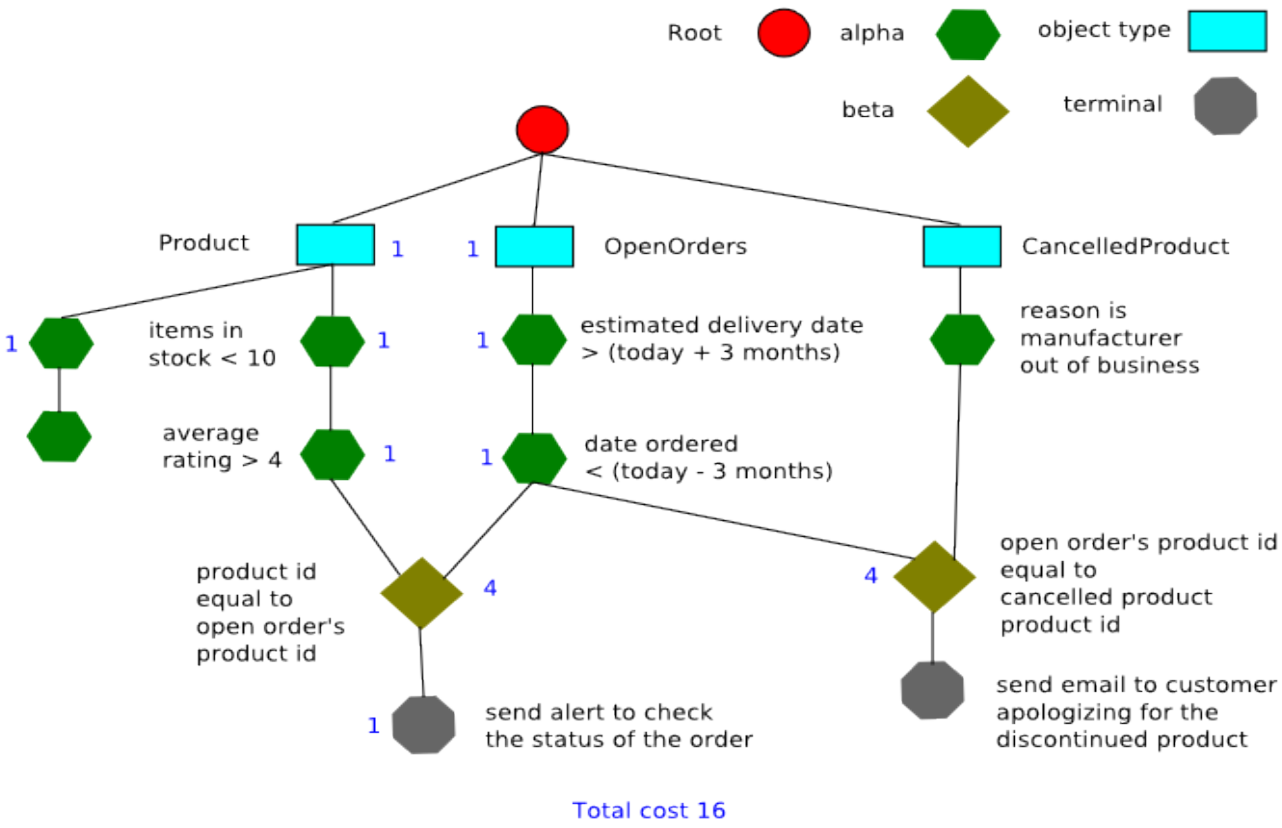
Cost Diagram 2: ruleset

The second example shows a network that has several rules. The two object type nodes have several children, which increases the cost for rule. This assumes the object type node will propagate the facts to all child nodes. One important thing to note is that even if the number of rules increase, the cost of the rule might not increase. Generally, the cost will increase only if the object type nodes used by the rule have more children. To get around that issue, some rule engines like Jamocha use object type node hashing.



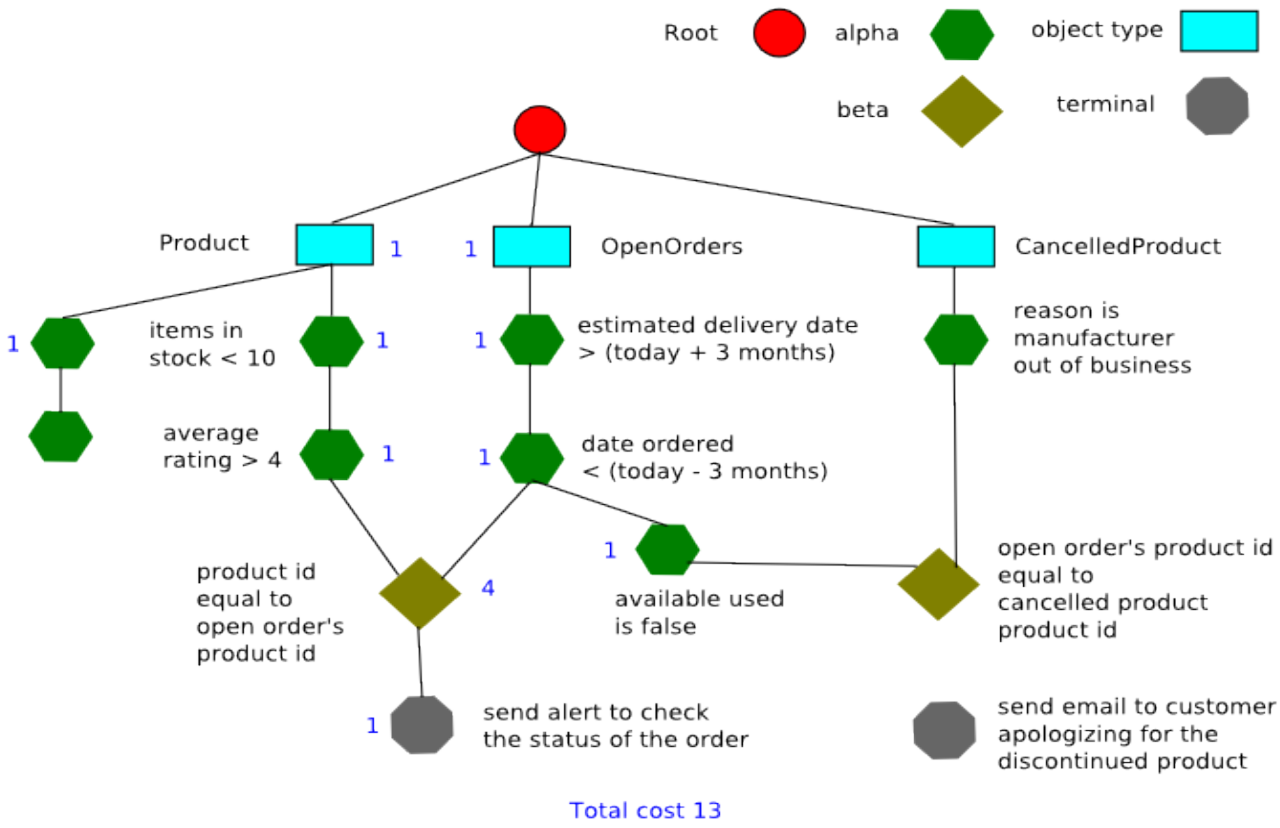
Cost Diagram 3: node sharing

In the third example, we have 2 rules that share the same conditions for OpenOrders. If we change “send alert to check the status of the order” rule so it has a third condition for OpenOrders, the cost for “send email to customer” rule would be 11 instead.



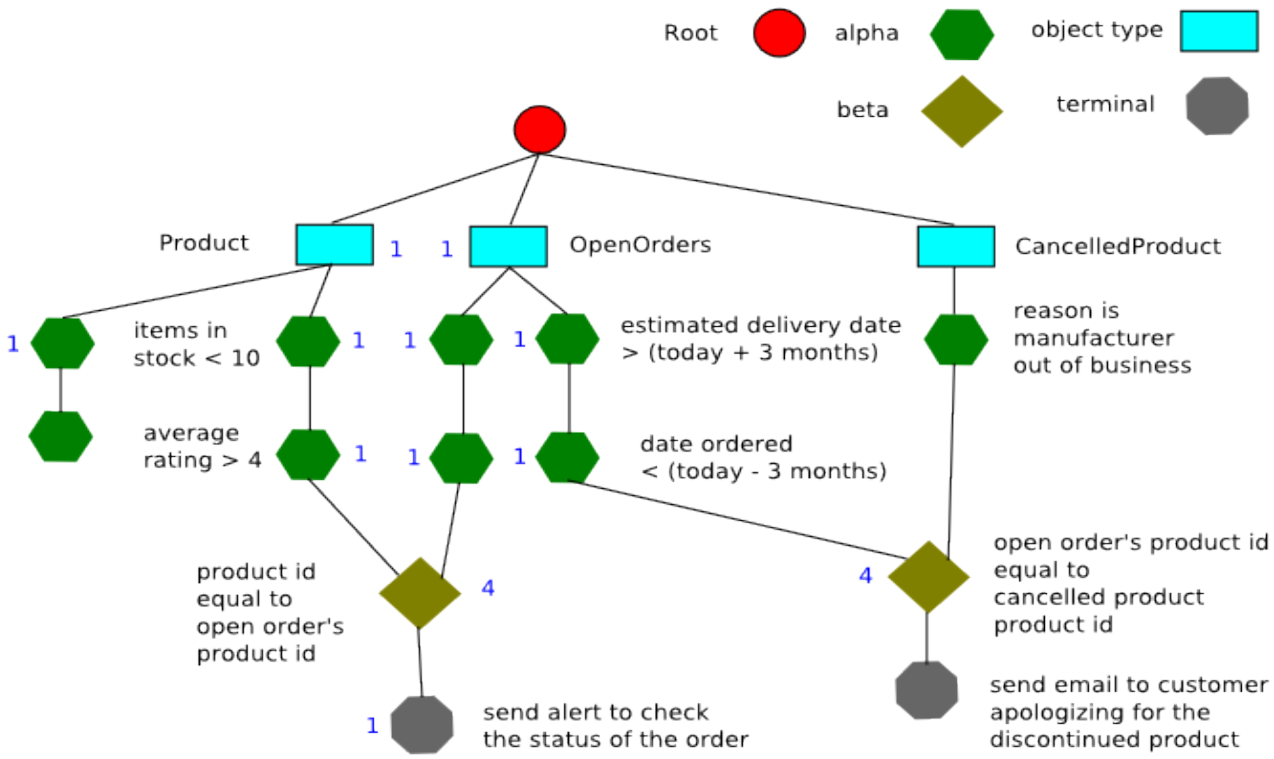
Cost Diagram 4

In the forth example, "send alert to check the status of the order" rule has a cost of 16. One might get the impression that alpha node sharing results in a higher cost. To get a better understanding, we can look at a slight variation.



Cost Diagram 5

Notice that if we add another literal constraint to "discontinued product" rule, the cost of "send alert to check the status of the order" rule goes down. From a real world perspective, the difference between a cost of 13 and 16 isn't significant for a good implementation. For a slow RETE implementation, that cost "could" add up and make a real difference. Here is what the cost would be if alpha node sharing isn't used.



Cost Diagram 6

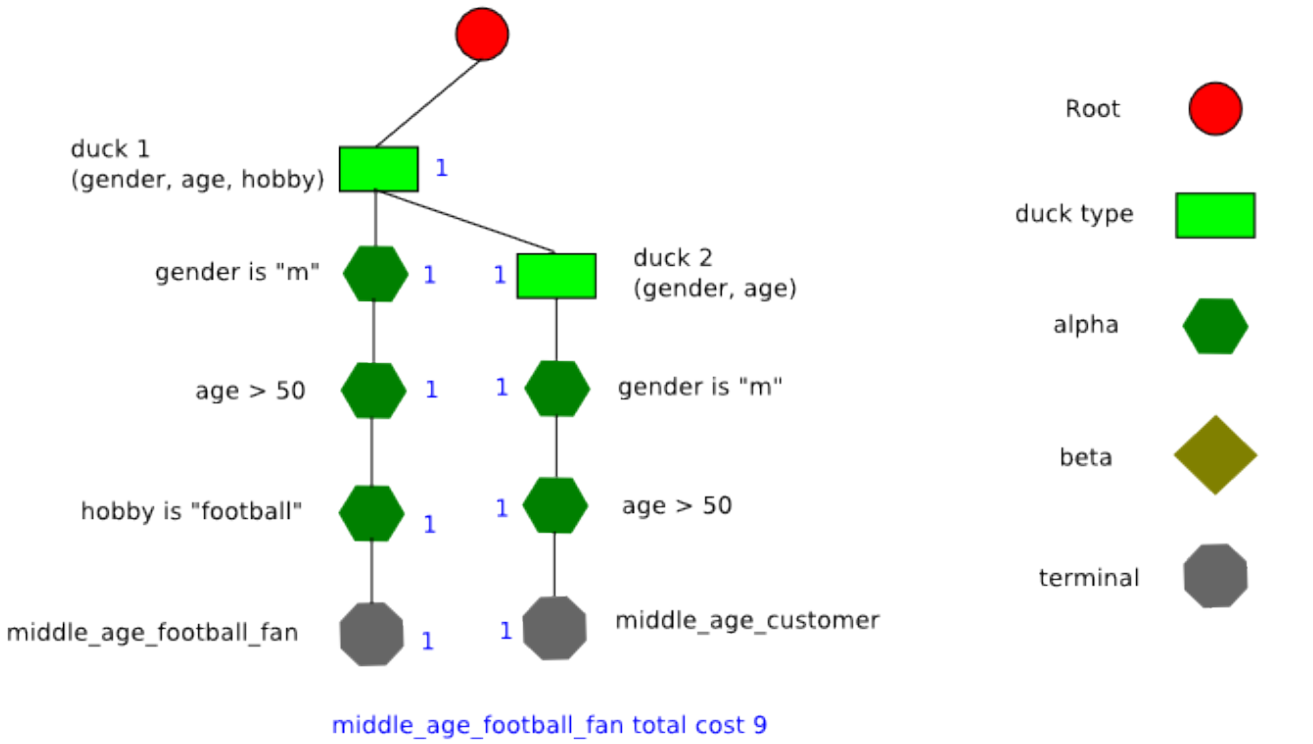
Notice the cost of “send alert to check the status of the order” rule goes from 16 to 18. Sharing the alpha nodes reduces the cost by 2. The basic concept is to look at how many nodes are visited in the process of evaluating a rule. To do that, first we compile the rules and generate the RETE network. Once we have the network, we look at which nodes would be visited if we assert facts for a rule. The function can be described by the following pseudo code.

```
int cost = 0

// first calculate the cost of the object-type node
foreach conditional element
      find the object type node

      // calculate the cost of the object-type node's children

      int child = 1
      foreach child of object type node
            if node is alpha node
                  child = child + 1
            else if node is beta node
                  child = child + 4
            else if node is Not or Exist
                  child = child + 5
            end
      end

      cost = cost + child

// second calculate the cost of the conditions for each conditional element
foreach conditional element

      get the children
      for int count=1; count < children.count; count++
```

```
            if node is alpha node
                  cost = cost + 1
            end
      end

end

// third calculate the cost of the join nodes
foreach join node in the rule
      cost = cost + 4

      if node is existential or negated

            cost = cost + 1

      end

end
```

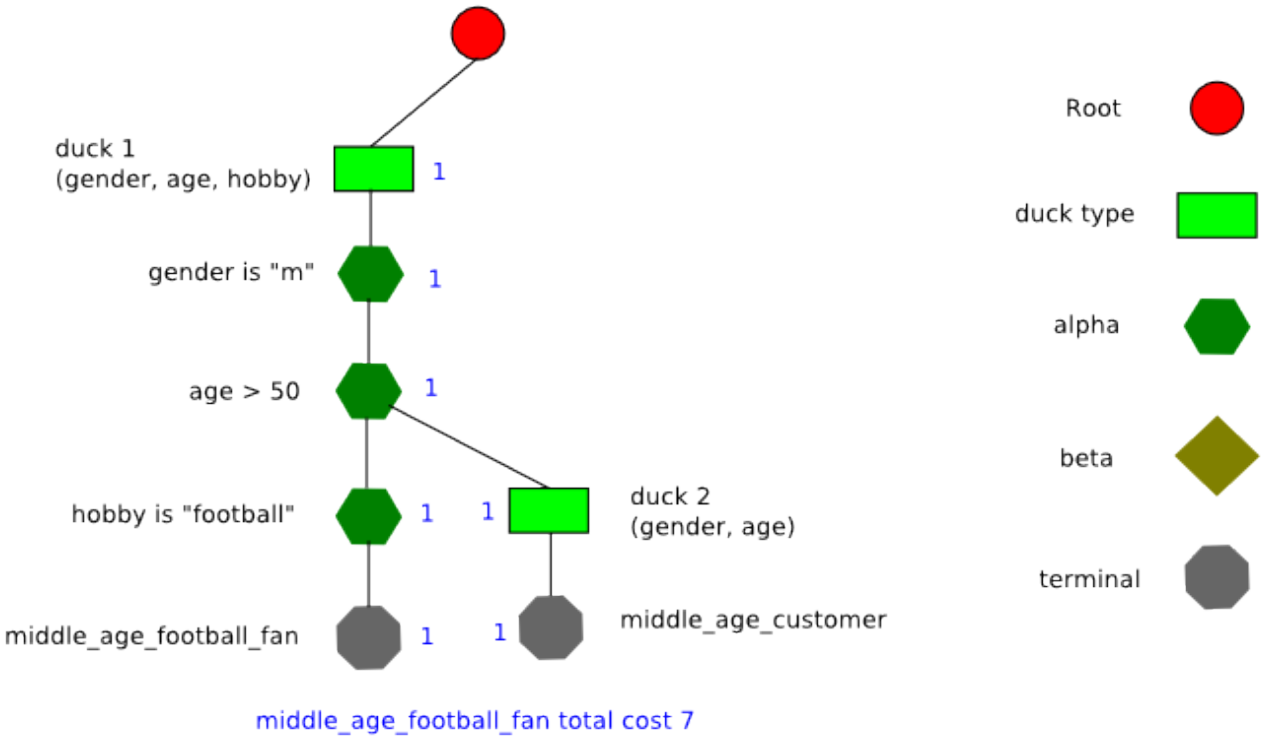
A couple of important notes about the cost function. It doesn't take into account engines that implement object-type node optimization. It also ignores implementation specific features like left-input adapter nodes. Engines that implement additional specialized nodes should alter the cost function appropriately. As the algorithm shows, each 1-input node has a value of 1. The join nodes have a value of 4 and exists and not have a value of 5. These values are based on the relative cost between 1 and 2 input nodes. Depending on the rule engine, the cost difference between an alpha and beta node might be greater.

We can apply the same type of cost calculation for duck-type RETE. Using the example duck-type RETE network, we can calculate the cost.



Duck-type Cost Diagram 1

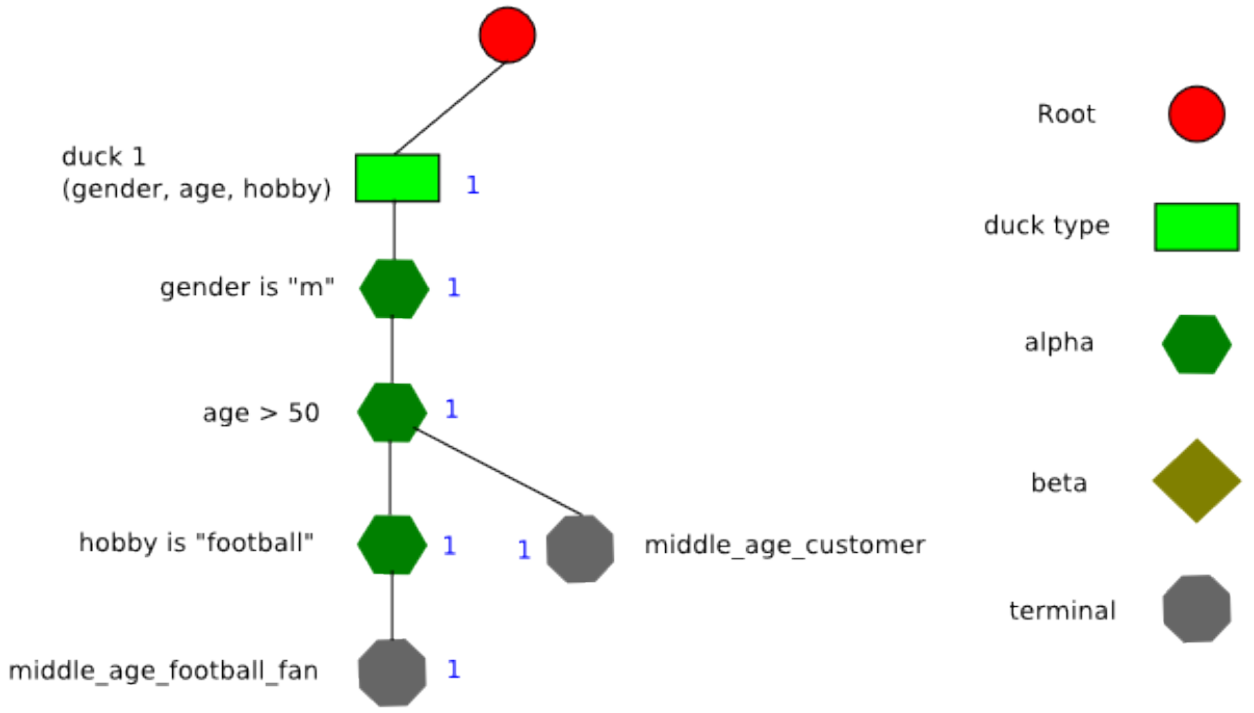
Note that “middle_age_customer” will always fire when “middle_age_football_fan” fires.



middle_age_football_fan total cost 7

Duck-type Cost Diagram 2

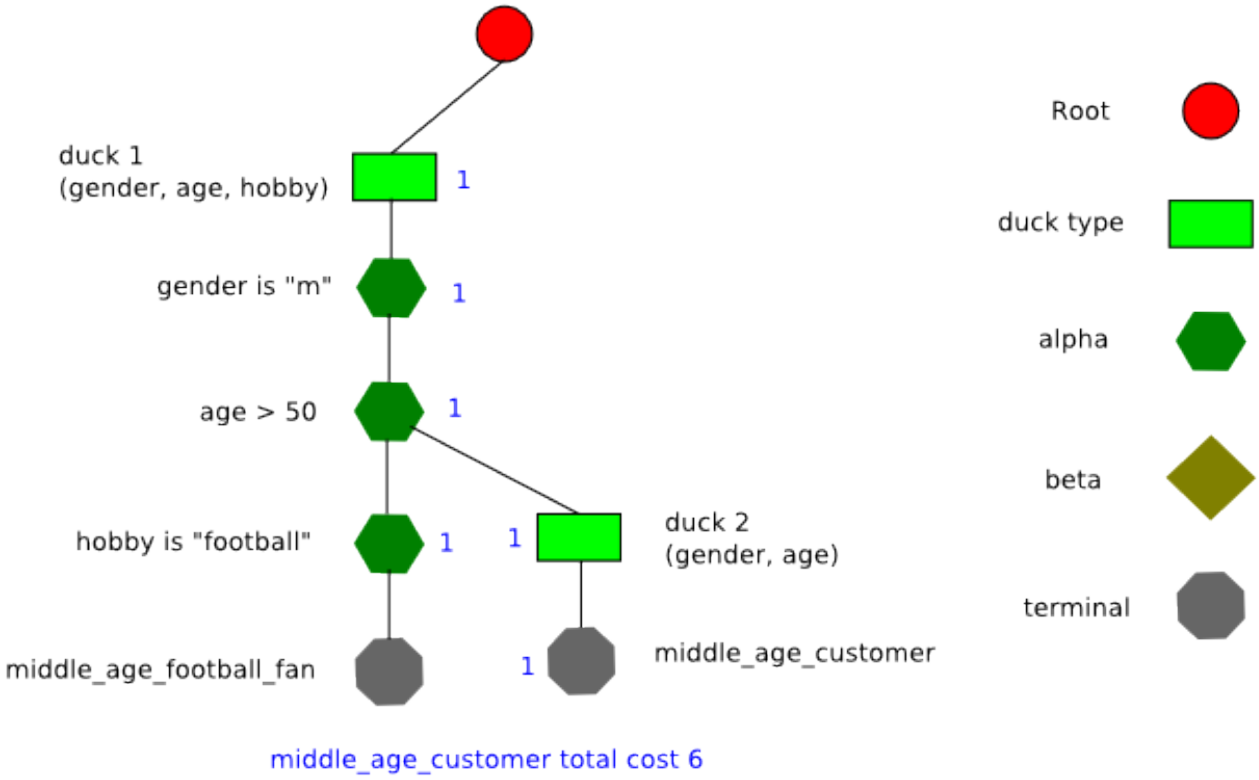
Note that sharing nodes reduces the cost by 2.



middle_age_football_fan total cost 6

Duck-type Cost Diagram 3

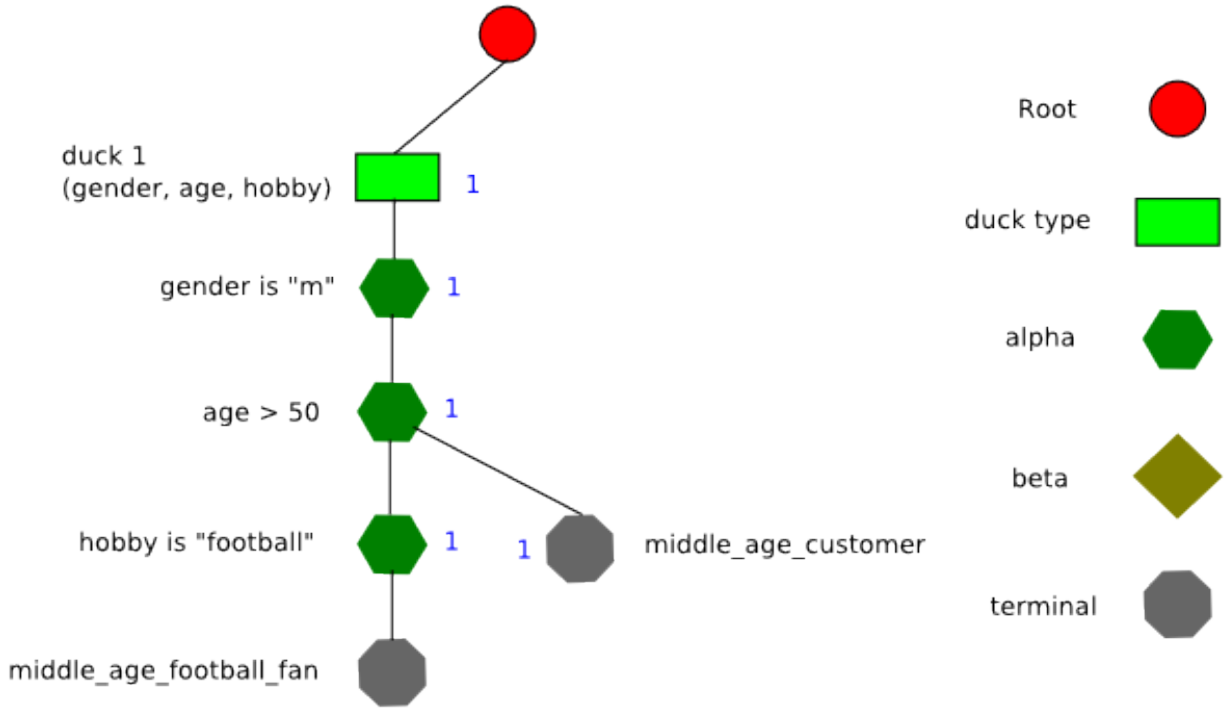
When we remove duck2 node, the cost drop by 1.



middle_age_customer total cost 6

Duck-type Cost Diagram 4

If we look at middle_age_customer, node sharing reduces the cost from 8 to 6.



middle_age_customer total cost 5

Duck-type Cost Diagram 5

Calculating the cost of duck-type rules is similar. First we compile the rules and generate the RETE network. After the rules are compiled, we look at which nodes are visited for each rule to calculate the cost. Once we have the cost for each rule, we can compare the cost between statically typed and dynamically typed. A simple comparison would be to divide the cost values

```
duck-type cost / object-type cost = relative weight
```

For example, if the statically typed rule has a cost of 10 and the equivalent duck-type rule is 18, the relative weight is 18/10 = 1.8. We can also reverse the formula and divide 10/18 = 5.555. Depending on the type of networks we want to compare, the formula can be adjusted. One thing to note is the relative weight does not mean one network will perform better by that factor. The actual benefit will depend on the implementation. One benefit of the topology cost function is we can compare the performance of two equivalent rulesets by averaging the cost.

1. Calculate relative weight for each business rule
2. sum (weight) / rule count = average weight

Conclusion

Although this paper explores some of the challenges and suggests potential implementation strategies, the larger question of dynamic typing RETE has not been fully addressed. Building a reliable duck typing and dynamic typing RETE rule engine requires more research. The first step is to identify the use cases and limitations. For the typical business use case, a duck type rule engine may not be useful. No data exists today that indicates a need for duck type RETE, but it could be an interesting strategy for machine learning or semantic web applications.

Another area for future research is the topology cost function. The cost function can be applied to a variety of pattern matching optimization problems. More specifically, a machine learning application could use the topology cost function to rate the cost of new rules before adding them. The cost function can also be enhanced to consider the cardinality of the data.